Zur Kontextanalyse einer algebraischen Programmiersprache

vorgelegt von
Diplom-Informatiker
Christian Maeder
aus Kevelaer

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Klaus Obermayer
Berichter: Prof. Dr. Peter Pepper
Berichter: Prof. Dr. Stefan Jähnichen

Tag der wissenschaftlichen Aussprache: 28.06.2001

Berlin 2001
D 83

Danksagung

Die vorliegende Arbeit entstand während und nach meiner Lehr- und Forschungstätigkeit als wissenschaftlicher Mitarbeiter im Fachgebiet „Übersetzerbau und Programmiersprachen“.

Dank gebührt vor allem meinem Doktorvater und dem Initiator des OPAL-Projekts Prof. Peter Pepper. Ohne seine Ermutigungen wäre diese Arbeit sicher nicht zustande gekommen. Viele ehemalige Mitarbeiter haben mich bei meinen Tätigkeiten unterstützt, für ein angenehmes Arbeitsklima gesorgt und zum OPAL-System beigetragen. Davon zu nennen sind Jürgen Exner, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, Thomas Nitsche und mein Vorgänger Michael Jatzeck. Mit Mario Südholt verband mich darüberhinaus ein gemeinsames Büro und das Interesse am Schachspiel. Bei Klaus Didrich möchte ich mich für seine langjährige Kontinuität bei der Wartung des OPAL-Compilers, seine inhaltlichen Ansichten und sein sorgfältiges Korrekturlesen bedanken.

Prof. Stefan Jähnichen danke ich für die Übernahme der Begutachtung dieser Arbeit sowie für eine Beschäftigung als GMD-Mitarbeiter.

Viel emotionalen Anteil an meiner Arbeit nahmen meine Freunde und Familie. In besonderer Weise schulde ich aber meiner Frau Christine Dank für ihre vielfältige Unterstützung und Geduld.

Berlin, Januar 2001 Christian Maeder

Zusammenfassung

Algebraische Spezifikationen wurden theoretisch gut untersucht, doch für verschiedene Entwürfe von konkreten Spezifikations- oder Programmiersprachen erwies sich die Notation von *Instanzen* für parametrisierte Strukturen als umständlich. Inzwischen hat sich dafür eine Listennotation durchgesetzt, bei der aktuelle und formale Parameter per Position zugeordnet werden. In einigen Sprachen wurden explizite Instanziierungen reduziert, indem parametrisierte Namen *generisch* importiert werden können und die *Kontextanalyse* die fehlende Information aus dem Anwendungskontext inferiert. Der *Sprachentwurf* orientiert sich insoweit an Analysealgorithmen. Der Entwurf von ML mit streng getypten polymorphen Funktionen höherer Ordnung wurde geprägt durch den Hindley-Milner-Algorithmus $\mathcal{W}$ zur Typinferenz. Der ursprüngliche Entwurf von PASCAL war durch die effiziente One-Pass-Technik auf so genannte *lineare Sichtbarkeit* beschränkt. Diese gilt immer noch für fast alle *algebraischen* Sprachen (PVS, LPG und CASL), aber nicht für moderne Programmiersprachen wie JAVA (objektorientiert) oder HASKELL (funktional). Der Entwurf von OPAL ist daher herausragend: Deklarationen können in beliebiger Reihenfolge notiert werden und sind im ganzen *Modul* sichtbar; unabhängig davon ist die Reihenfolge formaler Parameter.

Für die algebraische *Typanalyse* werden in dieser Arbeit die klassische polymorphe Typinferenz und *Überlagerungsauflösung* zu einem Algorithmus $\mathcal{W}_o$ verschmolzen. Es stellt sich heraus, dass dieselbe Art der Analyse für die *Identifikation* von überlagerten und generischen *Namen* benötigt wird. Dafür müssen einfache Typterme zu *Namenstermen* verallgemeinert werden; die Namen stehen für deklarierte Typen und Funktionen, die sowohl die Parametersignatur als auch die Gesamtsignatur einer Struktur etablieren. Durch den hier entwickelten Algorithmus $\mathcal{I}$ für die Namensidentifikation entsteht eine Parallele zur Typanalyse, die über die aus HASKELL bekannte Analogie für die Konstruktorapplikation von Typen und Ausdrücken hinausgeht. *Instanziierung* ist *Applikation* und zwar insbesondere für Funktionsparameter. Dadurch werden die klassische ML-Polymorphie und Funktionen höherer Ordnung zu Teilaspekten der universelleren Generizität durch Parametrisierung. Den Kern der *Namensidentifikation* $\mathcal{I}$ bilden die um Funktionen erweiterten *Namensterme*. Die konsequente Gleichberechtigung von Typen und Funktionen unterstützt Funktionsparameter, die andere Funktionsparameter enthalten können; diese sind beispielsweise für die Zusicherung von Eigenschaften geeignet. Der induktive Aufbau der Namen erlaubt die inkrementelle Konstruktion eines Namensraums unabhängig von der textuellen Reihenfolge. Für einen erweiterten und unverändert schlanken Sprachentwurf werden *implizite* Parameter vorgeschlagen, die Parameterlisten verkürzen, sowie sich wechselseitig importierende Strukturen und polymorphe Rekursion.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Kontextanalyse 1

1.1.1 Parametrische Polymorphie 2

1.1.2 Das Annotationskonzept 4

1.2 Generische Namen und Überlagerung 7

1.3 Algebraische Spezifikationen 9

1.3.1 Parametrisierung mit Funktionen 9

1.3.2 Instanziierung 11

1.3.3 Namensidentifikation 14

1.3.4 Namensräume 15

1.4 Historie und Motivation 16

1.4.1 Beweisbarkeit 16

1.4.2 Werkzeugintegration 18

1.5 Übersicht 19

2 Polymorphe Typinferenz **21**

2.1 Datenstrukturen 21

2.1.1 Ausdruck 21

2.1.2 Typ 23

2.1.3 Substitution 25

2.1.4 Typschema 26
2.1.5 Umgebung 27
2.2 Typableitung 28
2.3 Algorithmus $\mathcal{W}$ 30
2.3.1 Ausführliches Beispiel 33
2.3.2 LET-Ausdrücke 34
2.3.3 Komplexität 35
2.4 Unifikation 37
2.4.1 Algorithmus 37
2.4.2 Komplexität 39
2.5 Algorithmus $\mathcal{M}$ 40
2.6 Grenzen der polymorphen Typinferenz 43

3 Überlagerungsauflösung 45
3.1 Brute-Force Algorithmus 47
3.1.1 Beispiele 50
3.1.2 Varianten 51
3.2 Attributierungsalgorithmus 53
3.3 Erweiterungen 55
3.3.1 Inferenz von fehlender Typinformation 56
3.3.2 Lambda-Ausdrücke 56
3.3.3 Funktionen höherer Ordnung 57
3.4 Verschattung und Überlagerung 59

4 Algebraische Typanalyse 63
4.1 Analysealgorithmus $\mathcal{W}_o$ 65
4.2 Restriktion: Lokale LET-Eindeutigkeit 67

4.3 Verallgemeinerung: Überlagertes LET 69
4.4 Monomorphie 70

5 Namen und Instanzen **73**
5.1 Endliche Mengen 74
5.2 Abhängige formale Parameter 77
5.3 Vollständige Namen 79
5.3.1 Substitution 80
5.3.2 Unifikation 81
5.4 Partielle Namen 82
5.4.1 Redundante Typannotationen 83
5.4.2 Überlagerung innerhalb einer Struktur 84
5.4.3 Inferierbarkeit der Instanz 85
5.5 Funktionen höherer Ordnung 88

6 Namensidentifikation **91**
6.1 Identifikationsalgorithmus $\mathcal{I}$ 92
6.1.1 Inferenz der Instanz 93
6.1.2 Inferenz unbekannter Namen 94
6.1.3 Identifikation von Typannotationen 95
6.2 Interpretation der Ergebnisliste 95
6.2.1 Variablenfreiheit 96
6.2.2 Mehrdeutigkeit 96
6.3 Offener Namensraum 98
6.4 Monotonie der Identifikation 100

7 Import **101**
7.1 Eindeutige Importinstanz 103

7.1.1 Lokale versus globale Auflösbarkeit 105
7.1.2 Generische versus mehrfach instanziierte Namen 106
7.2 Selektiver Import 108
7.2.1 Selektiver Ausschluss 109
7.2.2 Trennung von Typen und Funktionen 110
7.2.3 Generische Instanz 111
7.3 Direkter Import 112
7.4 Transitiver Import 114
7.4.1 Herkunftsannotation 116
7.4.2 Instanzannotation 117
7.4.3 Reexport generischer Namen 118
7.4.4 Teilinstanziierte Namen 119
7.4.5 Namensraumexplosion 120
7.4.6 Abhängige Importinstanzen 121

8 Namensraumanalyse **125**
8.1 Reihenfolgeunabhängigkeit 127
8.1.1 Abschließender Mehrdeutigkeitstest 129
8.1.2 Eindeutiges Beispiel 130
8.1.3 Wiederholte Deklaration 131
8.2 Abgeschlossene Parametersignatur 134
8.3 Implizite Parameter 137
8.4 Modulkonzepte 140
8.4.1 Zyklische Strukturen 141
8.4.2 Modulhierarchie 144
8.4.3 Synonyme 145
8.4.4 Schnittstelle und Implementierung 146

9 Ergebnisse **147**

9.1 Algorithmus $\mathcal{I}$ 147

9.2 Namensraumkonstruktion 150

9.3 Verwandte Arbeiten 152

9.4 Zukünftige Arbeiten 153

Kapitel 1

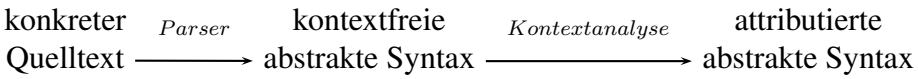
Einleitung

Algebraische Sprachen, meistens Spezifikationssprachen, wurden fast ausschließlich im Hinblick auf Semantik, *Requirements Engineering* und Beweisstrategien untersucht und entworfen. In dieser Arbeit wird speziell die Namensraum- und Kontextanalyse für eine funktional-algebraische *Programmier*sprache im Stil von Opal [Exn94, Gro94, DEGP94, DFG+94, DGG+96, Pep98] vorgestellt. Das konkrete Ziel ist eine beweisbar korrekte Implementierung, die ausreichend effizient und benutzerfreundlich ist. Besonders beleuchtet werden dabei der Zusammenhang zwischen bestimmten Sprachentwurfsentscheidungen und der Verkomplizierung der Kontextanalyse sowie die Unterschiede und Gemeinsamkeiten im Vergleich zu den klassischen und den neueren *funktionalen* Programmiersprachen mit strenger Typisierung.

Schlüsselwörter: Polymorphie, Funktionen höherer Ordnung, Typinferenz, Unifikation, Verifikation, Ad-hoc-Polymorphie, Überlagerungsauflösung, algebraische Spezifikation, Signatur, Parametrisierung, Namensidentifikation, Namensraum, Modularisierung

1.1 Kontextanalyse

Eine kontextfreie *abstrakte Syntax* [ASU86] ist der Ausgangspunkt der Kontextanalyse; den Endpunkt bilden eindeutige, benannte *semantische Entitäten*, in erster Linie die Typen und Funktionen. Die abstrakte Syntax wird dabei normiert und durch Typannotationen vervollständigt. Die wesentlichen dadurch aufgedeckten Fehler sind inkonsistent oder mehrdeutig verwendete *Namen*:



Die so genannte *Signaturanalyse* liefert den *Namensraum* für die anschließende *Typanalyse* von (definierenden) Funktionsausdrücken des erweiterten λ-Kalküls [Bar84, Bar91]. Signatur- und Typanalyse zusammen bilden die *Kontextanalyse*. In der Literatur zum Übersetzerbau [WG84, Jon87, WM92] wird die Kontextanalyse synonym auch als *statische* oder *semantische Analyse* bezeichnet:

$$\text{Signaturanalyse} \quad + \quad \text{Typanalyse} \quad = \quad \text{Kontextanalyse}$$

1.1.1 Parametrische Polymorphie

Der Begriff „funktional-algebraisch“ (oder [Gog84] *parametrische Programmierung*) beschreibt die Verschmelzung zweier Sprachparadigmen, die beide die *parametrische Polymorphie* [CW85] unterstützen. Auf der einen Seite stehen klassisch funktionale Programmiersprachen mit strenger Typisierung wie ML [DM82, Pau96]; auf der anderen Seite sind es algebraische Spezifikationen [EM85, EM90] mit einem Parametrisierungskonzept.

Für die *Programmierung im Großen* bieten algebraische Sprachen Theorien oder *Strukturen* als Module, die in einer *azyklischen Importrelation* stehen. Die Strukturen können mit Typen *und Funktionen parametrisiert* werden.

Die uniforme Polymorphie durch *Parametrisierung mit Typen* kann mit der bekannten ML-Polymorphie verglichen werden, z.B. am *marvelous sequence type*:[1]

```
STRUCTURE Seq[α]
  TYPE α                                  - formaler Typparameter
  TYPE seq = ◇                            - leere Liste
             ::(ft : α, rt : seq)         - Kopf und Restliste
  IMPORT Nat COMPLETELY
  FUN # : seq → nat                       - Listenlänge
  DEF #(S) = IF ◇?(S) THEN 0
             ELSE 1 + #(rt(S)) FI
```

Dasselbe sieht in ML wie folgt aus:

[1] In konkreter OPAL-Syntax werden die Schlüsselwörter SIGNATURE und SORT (außer beim freien Typ) verwendet. DEF-Gleichungen müssen im *Implementierungsteil* stehen.

```
datatype 'a list = nil | :: of 'a * 'a list ;

fun len(nil) = 0
  | len(_::R) = 1 + len(R) ;
```

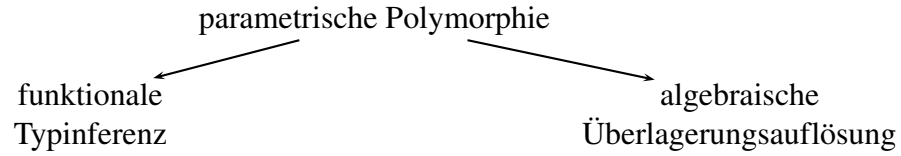
Die Hauptunterschiede sind syntaktischer Natur: die algebraische Notation ist länger aber durch redundante Typinformation expliziter. Semantisch sind die Unterschiede nur noch marginal und subtil. Ein Aspekt dieser Arbeit ist die detaillierte Diskussion der Unterschiede und ihre Überwindung im Hinblick auf eine engere funktional-algebraische Verschmelzung. Parametrisierte Strukturen sowie die darin enthaltenen Typen und Funktionen werden üblicherweise *instanziiert* und *importiert*:

```
STRUCTURE SeqMap[α, β]
  TYPE α
  TYPE β                                  - formale Parameter
  IMPORT Seq[α] COMPLETELY
  IMPORT Seq[β] COMPLETELY                - instanziierte Importe

  FUN map: (α → β) × seq[α] → seq[β]      - Funktionsdeklaration
  DEF map(F, S) =
      IF ◊?(S) THEN ◊
      ELSE F(ft(S)) :: map(F, rt(S)) FI   - definierender Ausdruck
```

Mehrere Instanzen, wie `Seq`$[\alpha]$ und `Seq`$[\beta]$, bewirken, dass die importierten Funktionen, $\Diamond$, $\Diamond?$, `ft`, `rt`, `::`, etc. *überlagert*[2] vorhanden sind; d.h. die bloßen Funktionssymbole (oder *Identifier*) bezeichnen *verschiedene* Funktionen nicht eindeutig.

Die parametrische Polymorphie wird funktional und algebraisch unterschiedlich betrachtet und analysiert. Im klassisch-funktionalen Stil sind die Funktionen über Sequenzen *polymorph* und der *prinzipale* (oder *allgemeinste*) Typ einer Funktion kann aus der Definitionsgleichung *inferiert* werden; die Angabe einer *Signatur*, die eine Funktion mit ihrem Typ explizit *deklariert*, ist dabei nicht nötig. Algebraisch werden statt *einer* polymorphen Funktion *mehrere monomorphe Instanzen* betrachtet; mehrfach instanziierte Importe liefern somit überlagerte Funktionen und entsprechend ist die Typanalyse *Überlagerungsauflösung*:

[2] In der deutschsprachigen Literatur wird der Begriff *Overloading* oft wörtlich als *Überladung* übersetzt; hier wird *Überlagerung* bevorzugt.

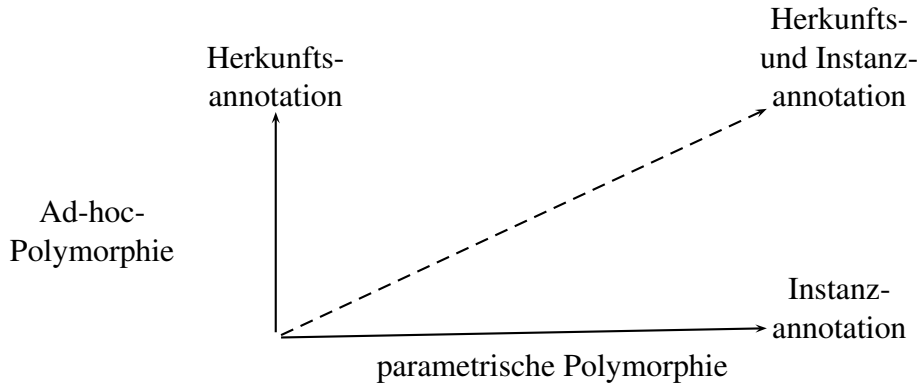


Überlagerungsauflösung basiert auf den unterschiedlichen Typen überlagerter Funktionen; dadurch *passt* in einem Ausdruck (dem Anwendungskontext) meistens genau eine Funktion typkorrekt. Sind überlagerte Funktionen typgleich oder ermöglicht ein Anwendungskontext ausnahmsweise keine eindeutige Auflösung, dann muss die Funktion vom Benutzer – dem Programmierer – *annotiert* werden. Die durch mehrere Instanzen überlagerten Funktionssymbole können dabei mit *Instanzannotationen* vervollständigt werden; *Bezeichner* und Annotation zusammen bilden dann einen eindeutigen *Namen.*

Im obigen `map`-Beispiel sind explizite Instanzannotationen, wie $\Diamond[\beta]$, $\Diamond?[\alpha]$, `ft`$[\alpha]$, `rt`$[\alpha]$ und `::`$[\beta]$, nicht nötig; die fehlenden Instanzen können automatisch ergänzt werden. Die mit α instanziierten *Selektoren zerlegen* dabei Sequenzen vom Typ `seq`$[\alpha]$ und die mit β instanziierten *Konstruktoren* erzeugen Sequenzen vom Typ `seq`$[\beta]$. Die Typen der instanziierten Funktionen (darin insbesondere die entsprechenden Instanzen zum Typkonstruktor `seq`) unterscheiden sich und ermöglichen eine eindeutige Überlagerungsauflösung.

1.1.2 Das Annotationskonzept

Die Unterstützung von allgemeiner Überlagerung, d.h. *Ad-hoc-Polymorphie*, ist geradezu ein Charakteristikum algebraischer Sprachen: Funktionssymbole dürfen beliebig wiederverwendet werden. Bei überlagerten Funktionen aus unterschiedlichen Strukturen entspricht die so genannte *Herkunftsannotation* der aus vielen anderen Sprachen bekannten *Namensqualifikation* mittels Punkt-Notation, durch die Namen global eindeutig werden:



In OPAL wird die Herkunft (*Origin*) mit einem Apostroph notiert; außerdem ist (aus Orthogonalitätsgründen) die Reihenfolge der Bezeichner in `ide'Mod` im Vergleich zur üblichen Punkt-Notation `Mod.ide` vertauscht. Zum Beispiel könnte man eine leere Sequenz ◊ wie folgt annotieren:

◊[`nat`]	- *Instanzannotation*
◊'`Seq`	- *Herkunftsannotation*
◊'`Seq`[`nat`]	- *beide Annotationen*
◊'`Seq`[`nat'Nat`]	- *Herkunft zur Instanz*

Statt *Annotation* wird synonym auch der Begriff *Qualifikation* verwendet. Herkunftsannotationen *vermeiden* Überlagerung, da qualifizierte Name eindeutig sind, wenn die Module global eindeutig sind. (Auch in OPAL müssen sich Strukturnamen unterscheiden.) Überlagerungsauflösung ist nur nötig, wenn Bezeichnungen nicht eindeutig sind. Ob und wie Überlagerung tatsächlich aufgelöst wird, hängt von der Sprache ab, ist aber unabhängig vom Sprachparadigma (imperativ, objektorientiert, funktional oder logisch).

Namensqualifikation wird beispielsweise von HASKELL [Tho96, PH+97] unterstützt. Wie in der imperativen Sprache MODULA [Wir85] wird Überlagerung schon *vor* der Typanalyse ausgeschlossen. In den objektorientierten Sprachen JAVA und C++ ist nur *argumentseitige Überlagerung* erlaubt. Die Anzahl und die Typen *der Argumente* überlagerter Funktionen müssen sich (unabhängig vom Ergebnistyp) unterscheiden, damit eine einfache und effiziente (bottom-up) Überlagerungsauflösung möglich ist. Lediglich die Überlagerungsauflösung für ADA [WS80] ist mit der von algebraischen Sprachen vergleichbar.

Sollte eine Auflösung überlagerter Funktionen in einem Anwendungskontext nicht möglich sein, dann können durch *Annotationen* Mehrdeutigkeiten reduziert werden; umgekehrt minimiert (eine optimale) Überlagerungsauflösung die Notwendigkeit solcher Annotationen. Außer Herkunfts- und Instanzannotationen gibt es in OPAL noch *Typannotationen*: die Konstante für die leere Liste kann mit ihrem Typ durch ◊ : `seq` notiert werden; um Typen und (die bisher noch nicht betrachteten) Funktionen in Instanziierungen zu unterscheiden, kann man Typen mit (dem speziellen *Typ*[3]) TYPE annotieren.

Ein *Bezeichner* mit oder ohne *Annotationen* ist in der Regel nur ein *partieller Name* und potenziell mehrdeutig. Ein Name ist erst dann *vollständig*, wenn sämtliche Annotationen bekannt sind; insbesondere müssen dafür die Namen *innerhalb* von (Typ- und Instanz-) Annotationen vollständig bekannt sein. Erst ein *vollständiger Name* ist *global eindeutig*!

[3]Der übliche englische Begriff *Kind* wird hier als *Typ* übersetzt.

Ein vollständiger Name, den man natürlich nie explizit so angeben wird, sondern von der Kontextanalyse ermitteln lässt, sähe wie folgt aus:

◊′Seq[nat′Nat : TYPE] : seq′Seq[nat′Nat : TYPE] : TYPE

Die beträchtliche Länge vollständiger Namen – das Beispiel ist lediglich eine Konstante – ist technisch kein Problem: die Instanz Seq[nat′Nat : TYPE] zum Bezeichner ◊ und zum Typ seq ist identisch. TYPE ganz am Ende ist auch redundant, da rechts vom Doppelpunkt *immer* ein Typ steht. Sämtliche partiellen Namen ergeben sich einfach durch *Weglassen* von Annotationen; die Zeichen „′ “ und „ : “ leiten eine Herkunfts- bzw. Typannotation ein und die eckige Klammerung „[. . .]“ umfasst eine Instanzannotation. Der entscheidende Bezeichner, hier das Funktionssymbol ◊, steht immer ganz vorne; und das Gleiche gilt für die Typbezeichnungen nat und seq *innerhalb* der Annotationen. Es folgen einige Beispiele partieller Namen:

◊ : seq	– *einfache Typannotation*
◊ : seq : TYPE	– *redundante Typannotation*
◊[nat] : seq	– *Instanz- und Typannotation*
◊ : seq[nat]	– *instanziierte Typannotation*
◊′Seq : seq[nat]	– *mit Herkunftsannotation*
◊′Seq[nat] : seq′Seq	– *usw.*
◊′Seq[nat′Nat] : seq′Seq[nat : TYPE]	

Annotationen legen die folgende abstrakte Spezifikation der Kontextanalyse nahe:

1. Ein *vollständig* annotierter Name ist global eindeutig.
2. Namen im Programmtext sind nur *partiell* gegeben.
3. Die Kontextanalyse muss die partiellen Namen *typkonform* vervollständigen oder einen Fehler melden, falls das nicht *eindeutig* möglich ist.

Diese informelle Spezifikation war in [DGMP97] der Ausgangspunkt, einen Algorithmus als Verfeinerung der *globalen Suche* zu formulieren. Diese Idee wird vom KIDS-Entwicklungswerkzeug [Smi90] unterstützt, das an anderen Stellen, z.B. bei Scheduling-Problemen [SP93], erfolgreich eingesetzt wurde. Die Effizienz der globalen Suche basiert auf *Constraint Propagation* [PS96], d.h. auf (Konsistenz-) Bedingungen, die den initial großen Suchraum schrittweise beschneiden.

Dieser interessante und anspruchsvolle Ansatz wird hier nicht weiter verfolgt, sondern eine einfachere Namensvervollständigung bzw. Namensidentifikation in Kapitel 6 durch die Verallgemeinerung des klassischen Hindley-Milner-Algorithmus für die polymorphe Typinferenz [Mil78] angestrebt.

1.2 Generische Namen und Überlagerung

Eine Verkomplizierung für die Kontextanalyse bedeuten *uninstanziiert* (bzw. synonym *generisch* oder *polymorph*) importierte Namen. Ein generischer Import kann mehrere instanziierte Importe ersetzen, verkürzt also die Notation. Die bisher unterschätzte Zusatzaufgabe für die Kontextanalyse besteht aber darin, die tatsächlich benötigten Instanzen – dieses sind nur endlich viele – dem Programmtext bzw. den *Applikationsstellen* zu entnehmen.

In OPAL könnten in der Struktur `SeqMap` (aus Abschnitt 1.1.1) die beiden instanziierten Importe ohne weitere Änderungen durch folgenden *generischen* Import ersetzt werden:

```
IMPORT Seq COMPLETELY
```

Generische Namen verhindern die direkte Verwendung eines Standardalgorithmus zur Überlagerungsauflösung [WS80, ASU86, MF91, WM92], da dafür *vorher* endlich viele Instanzen zur Überprüfung bekannt sein müssten. Für potenziell unendlich viele Instanzen sind diese (in Kapitel 3 diskutierten) Auflösungsalgorithmen nicht geeignet.

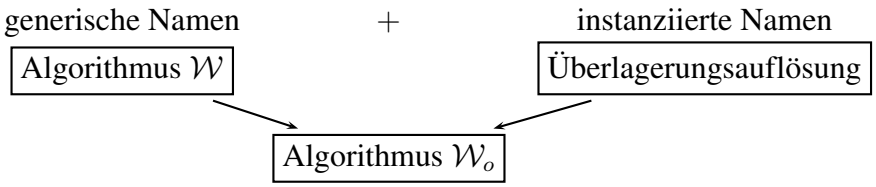
Das Herausfinden von Instanzen bzw. die Typanalyse mit generischen (also polymorphen) Namen legt nun einerseits die Verwendung des bekannten Hindley-Milner-Algorithmus $\mathcal{W}$ [Mil78, DM82, NN99] nahe: die *Spezialisierungen* für *Typvariablen* polymorpher Funktionen, entsprechen algebraisch den *Instanziierungen* von *Typparametern*. Andererseits unterstützt dieser Algorithmus nur die *parametrische* nicht aber die Ad-hoc-Polymorphie: in ML muss jede Funktion eindeutig benannt sein, lediglich ihr *Typ* darf polymorph sein. Der polymorphe Typ wird dabei durch *gebundene* Typvariablen repräsentiert.

Das *Binden* (oder *Generalisieren*) der Typvariablen auf einer äußeren Ebene charakterisiert genau die *shallow* Polymorphie für den Algorithmus $\mathcal{W}$ und die damit zuerst assoziierte funktionale Sprache ML. Der eng mit der Typinferenz verknüpfte Entwurf von ML gilt zu Recht als ein Meilenstein für Programmiersprachen und ist ein Beispiel für den wichtigen Einfluss ei-

nes Analysealgorithmus auf den Sprachentwurf. Die polymorphe Typinferenz wird ausführlich in Kapitel 2 rekapituliert.

Würde man die Einschränkung „keine Überlagerung“ einfach auf algebraische Sprachen übertragen, dann dürften dort alle Namen nur noch generisch importiert werden; dieses wäre zusammen mit der Beschränkung der Parametrisierung auf Typen eine durchaus brauchbare, mit ML vergleichbare Sprachvariante und eine Teilsprache von OPAL, die eine eigenständige Rolle spielen könnte. (Für unterschiedlich parametrisierte Funktionen werden natürlich mehrere Strukturen benötigt.)

Eine befriedigende Analyse von *generischen und instanziierten* Namen bietet die hier betrachtete Erweiterung des Hindley-Milner-Algorithmus um Ad-hoc-Polymorphie. Ein solcher Algorithmus wird im folgenden mit $\mathcal{W}_o$ bezeichnet (und in Kapitel 4 angegeben), wobei der Index o für *Overloading* steht.



Die Integration von Überlagerung und sogar Subtyp-Polymorphie[4] mit der parametrischen ML-Polymorphie ist nicht neu und wurde z.B. in [Smi91] vorgeschlagen. Praktisch hatte dieser Algorithmus aber keinen Einfluss auf den Sprachentwurf einer (funktionalen) Sprache; verbreiteter ist das *Typklassenkonzept* von HASKELL, mit dem Ad-hoc-Polymorphie gemäß [WB89] „weniger ad hoc“ integriert wurde. Dabei werden ad-hoc polymorphe Funktionen zu *einer*, semantisch nicht mehr *uniform* definierten, parametrisch polymorphen Funktion zusammengefasst.

Dem Benutzer muss klar sein, dass Überlagerung mit Bedacht und nicht willkürlich verwendet werden sollte. Entsprechende Warnungen gelten auch für andere Sprachen unabhängig von der Art und Effizienz der Überlagerungsauflösung.

Überlagerung vermindert die *Typredundanz* und kann die Les- und Wartbarkeit eines Quelltextes sowohl verbessern als auch verschlechtern. Mit geeigneten Annotationen wird die Sicherheit durch Redundanz wieder erhöht. Eine gewisse Rolle dafür spielen auch die deklarierenden Typsignaturen, in

[4]Mit Subtyp-Polymorphie wird die Typanalyse unentscheidbar.

OPAL sind das die durch FUN eingeleiteten Funktionsdeklarationen, die indirekt *Typannotationen* für definierende Ausdrücken darstellen. Diese Typsignaturen sind häufig an Stelle von ansonsten näherliegenden Instanz- oder Herkunftsannotationen ausreichend.

In den funktionalen Sprachen ist die Angabe einer Typsignatur *optional* und in den algebraischen *obligatorisch*. Insgesamt sind Signaturen für die Typ- und Überlagerungsauflösung unnötig, aber pragmatisch und methodisch äußerst sinnvoll; eine restriktive Entwurfsentscheidung für obligatorische Signaturen ist kaum nachteilig.

1.3 Algebraische Spezifikationen

Eine algebraische Spezifikation wird formal durch ein Tupel $\langle \text{S}, \text{OP}, \text{F} \rangle$ beschrieben, dabei sind S und OP Mengen von Symbolen für *Sorten* und *Operationen*. Für alle Operationssymbole aus OP müssen die Stelligkeiten sowie die Ein- und Ausgabesorten (also Typen) angegeben werden. Die Sorten und Operationen bilden dann *die Signatur* der algebraischen Spezifikation.

Semantisch sind die Operationen reine mathematische Funktionen, die nicht notwendig maschinell ausführbar sein müssen und z.B. Prädikate sein können. Den eigentlichen Inhalt der Spezifikation bilden logische Formeln F; einige (oder alle) davon entsprechen genau den definierenden Gleichungen eines funktionalen Programms.

In OPAL kann eine Signatur $\langle \text{S}, \text{OP} \rangle$ durch die Typ- und Funktionsdeklarationen mit Hilfe der Schlüsselwörter TYPE und FUN angegeben werden. Definierende Gleichungen für die Funktionen und andere spezifizierende Formeln werden durch DEF bzw. LAW eingeleitet. (In PVS [OSR93a] ist die Funktionsnotation PASCAL-ähnlich: `f(x: nat): nat = ...`)

1.3.1 Parametrisierung mit Funktionen

Spezifikationen können hierarchisch mit Hilfe einer azyklischen Importrelation *modularisiert* werden. Importierte Spezifikationen sind damit Teilspezifikationen und insbesondere ihre Signaturen sind Teile der Gesamtsignatur. Die Eigenschaften einer (importierbaren) Spezifikation werden getrennt angegeben und stehen damit für *übergeordnete* (importierende) Spezifikationen *unveränderbar* fest.

Im Vergleich zur parametrischen Polymorphie (aus Abschnitt 1.1.1) bei klassisch funktionalen Sprachen kommt bei den algebraischen Sprachen zusätzlich die Möglichkeit der *Parametrisierung mit Funktionen* hinzu. Die syntaktische Parametrisierung mit Typen *und Funktionen* ist semantisch als Parametrisierung mit einer – nicht importierbaren – Teilspezifikation zu verstehen. Diese formale Parameterspezifikation ist ein Platzhalter für *passende* aktuelle Spezifikationen. Die parametrisierte Spezifikation kann auf unterschiedliche *aber nur passende* Weise durch eine *aktuelle* Parameterspezifikation *instanziiert* werden; der aktuelle Parameter ist damit – vergleichbar mit einem Import – eine *Teilspezifikation* der instanziierten Spezifikation.

Die Parametrisierung in OBJ [FGMO87, GW88, GWM+93] erfordert die explizite Angabe von Theorien und Views: eine Theorie beschreibt eine (i.A. lose) formale Parameterspezifikation, ein View beschreibt die Korrespondenz (*Mapping*, *Morphismus*) zwischen einer Theorie und einer konkreten Spezifikation, die dann als aktuelle Parameterspezifikation geeignet ist. In OPAL und PVS [OSR93a] werden formale und aktuelle Typ- und Funktionsparameter direkt durch Parameterlisten (in eckigen Klammern) notiert.

Eigenschaften von formalen Parametern, wie `totalOrder`$[\alpha, <]$, bilden *Beweisverpflichtungen* für die Korrektheit einer Instanziierung. In PVS werden solche Parametereigenschaften durch Annahmen (*Assumptions*) angegeben, in OBJ sind sie Teile der Theorien.

```
STRUCTURE SeqOrd[α, <]
  TYPE α                                  - Elementtyp
  FUN <: α × α → bool                     - Elementordnung
  LAW totalOrder[α, <]                    - Parametereigenschaft
  IMPORT Seq[α] COMPLETELY                - instanziierter Import
  FUN <: seq × seq → bool                 - Sequenzordnung
  DEF <(s, S) =
      IF ◊?(S) THEN false
      ELSE IF ◊?(s) THEN true
      ELSE IF ft(S) < ft(s) THEN false
      ELSE IF ft(s) < ft(S) THEN true
      ELSE rt(s) < rt(S) FI FI FI FI
  LAW totalOrder[seq, <]                  - Eigenschaft
```

In OPAL unterscheiden sich Parametereigenschaften syntaktisch nicht von anderen Formeln; eine Unterscheidung ist nur implizit möglich: Parametereigenschaften sind Formeln, in denen nur Parameter (und höchstens von Parametern abhängige Importe) vorkommen. Die deklarierten (und definier-

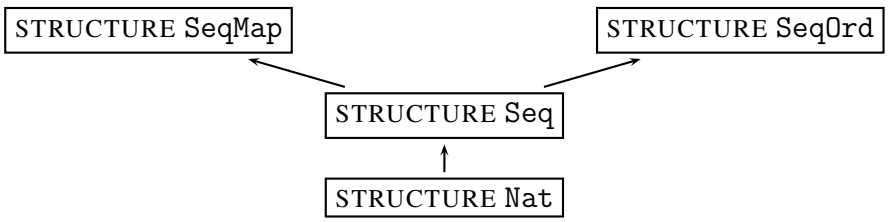
ten) Namen einer Struktur, im obigen Beispiel die Ordnung $<'$`SeqOrd` über dem Typ `seq`, ergeben sich schließlich in *Abhängigkeit von den Parametern* (und weiteren Importen). Die Formel `totalOrder`$[$`seq`$, <]$ beschreibt eine Eigenschaft der Sequenzordnung (und nicht der Parameter), da dem überlagerten $<$-Symbol in der Instanziierung von `totalOrder` eindeutig der Typ `seq` $\times$ `seq` $\rightarrow$ `bool` zugeordnet wird. (In der formalen Parameterliste `SeqOrd`$[\alpha, <]$ ist $<$ eindeutig, weil formale Parameter *nicht definiert* sein dürfen!)

Für die bloße Ausführung als Programm werden sämtliche Formeln, die in OPAL mit LAW eingeführt werden, ignoriert und als richtig vorausgesetzt. Die Kontextanalyse gewährleistet nur die Typkorrektheit einer Formel nicht aber ihre Gültigkeit. Für das Beweisen von Eigenschaften wird ein *Beweiswerkzeug* benötigt, wie es z.B. für PVS gegeben ist.

Dass die Sequenzordnung $<'$`SeqOrd` tatsächlich eine *totale* Ordnung ist, muss aus ihrer Definition folgen. Die Parameterspezifikation stellt sicher, dass eine totale Ordnung für die Elemente vorausgesetzt werden kann. Durch einen *instanziierten Import* von `SeqOrd` erhält man eine totale Sequenzordnung, die ihrerseits zur Instanziierung benutzt werden kann und dann eine totale Ordnung über Sequenzen von Sequenzen liefert. Dieses wird im folgenden Abschnitt 1.3.2 noch ausführlicher beschrieben.

1.3.2 Instanziierung

Ein grundlegender Teil der Kontextanalyse von algebraischen Sprachen ist die Bestimmung der *Signatur* $\langle$S, OP$\rangle$ einer durch Instanziierungen und Importe modularisierten Spezifikation $\langle$S, OP, F$\rangle$. Die Operationssymbole und ihre Typisierungen etablieren den *Namensraum* für die anschließende Typanalyse der Definitionen und Formeln. Importierte Strukturen werden getrennt und einmalig *vorher* analysiert. Die Importrelation der bisher vorgestellten Strukturen sieht graphisch folgendermaßen aus:



Parametrisierte Strukturen werden nur *einmal* analysiert; insbesondere die Typprüfung von definierenden Gleichungen und die spätere Codegenerierung[5] ist nur einmal nötig. Eine *passende* (und semantisch korrekte) Instanziierung garantiert, dass definierende Gleichungen typkorrekt (und Formeln semantisch korrekt!) bleiben.

Für die Signaturanalyse spielt die *Semantik* keine direkte Rolle und auch die nachfolgende Typanalyse prüft die Ausdrücke und Formeln lediglich auf Typkonsistenz. Die ganze Kontextanalyse ist unabhängig von der zu Grunde liegenden Logik, dem Korrektheitsbegriff und der Auswertungsstrategie zur Laufzeit des Programms. PVS [OSR93b, OS97b] verwendet eine zweiwertige, mengenbasierte Logik höherer Ordnung mit totalen Funktionen, totale Korrektheit und strikte Auswertung. Im Gegensatz zu OPAL kann mit PVS kein ausführbares Programm erzeugt werden, während für OPAL wiederum die Beweisunterstützung nur rudimentär ist.

Dass die Signaturanalyse bisher vernachlässigt wurde, zeigen allein schon die unterschiedlichen Sprachentwürfe für die algebraische *Instanziierung*. Beim *Signature Matching* von SML [MTH90, Pau96] bzw. im vergleichbaren Modulsystem von Leroy [Ler00] müssen zuerst die korrespondierenden Typen und Funktionen in formalen und aktuellen Parameterstrukturen *gleich bezeichnet* sein. Die statische Prüfung für die flexibleren Parameterlisten ist komplizierter und erfordert eine (im Vergleich zu ML) erweiterte Unifikation. Statt einer Unifikation mit *Typvariablen* ist algebraisch eine Unifikation mit *getypten Variablen* nötig. Der Unifikationsalgorithmus selbst muss dazu nicht verändert werden, lediglich die zu unifizierenden *Terme* unterscheiden sich: Typterme werden zu Namenstermen verallgemeinert.

Ein formaler Funktionsparameter, etwa $<$ aus der Struktur `SeqOrd` (aus Abschnitt 1.3.1), der beliebig benannt und (wie in `SeqOrd`) auch überlagert werden darf, ist eine *getypte Variable*. Der Typ dieser Variablen, $\alpha \times \alpha \to$ `bool`, enthält in diesem Beispiel den ersten formalen Parameter α. Der formale Typparameter α selbst ist auch eine getypte Variable und zwar mit dem speziellen *Typ* TYPE.

Die Instanziierung einer Struktur mit aktuellen Parametern ist insgesamt eine *typkonforme* Substitution von Variablen. Zum Beispiel könnten in einer weiteren (übergeordneten) Struktur die folgenden Importe notiert werden:

[5] Für jede Instanz speziellen Code zu erzeugen, wäre eine Optimierungsoption.

IMPORT `Nat` COMPLETELY
IMPORT `Seq`[`nat`] COMPLETELY
IMPORT `SeqOrd`[`nat`, <] COMPLETELY
IMPORT `SeqOrd`[`seq`[`nat`], <] COMPLETELY

Die beiden aktuellen Parameter in der Instanz `SeqOrd`[`nat`, <] stammen aus der Struktur `Nat`, wobei die Funktion <′`Nat` vom Typ `nat` × `nat` → `bool` ist. Dass der *formale* Parameter aus `SeqOrd` (zufällig) ebenfalls < heißt, ist belanglos. Der formale Parameter ist außerhalb von `SeqOrd` nicht sichtbar und eine Überlagerung damit nicht möglich. Es wird nur geprüft, ob der aktuelle Parameter <′`Nat` bezüglich seiner Typisierung – nicht bezüglich seiner Bezeichnung – zum formalen Parameter *passt*. Der aktuelle Typparameter `nat` in der Instanzliste [`nat`, <] determiniert (via α aus der formalen Parameterliste von `SeqOrd`) den Typ des aktuellen Funktionsparameter < zu `nat` × `nat` → `bool`.

Aus `SeqOrd`[`nat`, <] wird eine zweite Funktion < importiert; ihr (mit Herkunft und Instanz) annotierter Name lautet <′`SeqOrd`[`nat`, <], ihr Typ ist `seq`[`nat`] × `seq`[`nat`] → `bool` und damit passt sie als aktueller Funktionsparameter zur Instanz `SeqOrd`[`seq`[`nat`], <]. Aus `SeqOrd`[`seq`[`nat`], <] wiederum wird eine dritte Ordnungsfunktion < für Sequenzen von Sequenzen über natürlichen Zahlen `seq`[`seq`[`nat`]] importiert. Der obige Namensraum umfasst also mindestens die folgenden (ausführlich partiell annotierten) Namen:

`nat′Nat` : TYPE
<′`Nat` : `nat` × `nat` → `bool`
`seq′Seq`[`nat`] : TYPE
<′`SeqOrd`[`nat`, <′`Nat`] : `seq`[`nat`] × `seq`[`nat`] → `bool`
<′`SeqOrd`[`seq`[`nat`], <′`SeqOrd`[`nat`, <′`Nat`]]
: `seq`[`seq`[`nat`]] × `seq`[`seq`[`nat`]] → `bool`

Algebraisch sind Typen und Funktionen *gleichberechtigt*: sie sind die Hauptbestandteile von Strukturen und sie können formale Parameter sein. Im einfachen Fall (wie z.B. für die homogenen Listen der Struktur `Seq` aus Abschnitt 1.1.1) gibt es nur (einen) Typparameter. Die Funktion <′`SeqOrd` ist (zusätzlich) mit einer Funktion parametrisiert und vergleichbar mit einer Funktion höherer Ordnung. In Kapitel 5 wird der Typ `set` für endliche Mengen als Beispiel für einen mit einer Ordnungsfunktion parametrisierten Datentyp vorgestellt.

1.3.3 Namensidentifikation

Wie Funktionen können auch Typen *überlagert* sein. Dies ist z.B. für den Typ `seq` in der Struktur `SeqMap` (aus Abschnitt 1.1.1) der Fall: eine *Instanzannotation* ist nötig, um die *überlagerten* Typen `seq`$[\alpha]$ und `seq`$[\beta]$ auseinanderzuhalten. Die Instanzannotationen der Typen entsprechen dabei genau den üblichen Typkonstruktorapplikationen funktionaler Sprachen.

In der obigen Struktur `SeqOrd` (aus Abschnitt 1.3.1) dagegen wird nur *eine* Instanz `Seq`$[\alpha]$ importiert: der einfache Typname `seq` ist damit eindeutig `seq`$[\alpha]$. Auch können Typen aus verschiedenen Strukturen überlagert werden; dann helfen entsprechende Herkunftsannotationen wie z.B. `t'A` für den folgenden überlagerten Typ `t`:

```
IMPORT A ONLY t : TYPE
IMPORT B ONLY t : TYPE
FUN < : t'A × t'A → bool
```

Unifikation alleine reicht nicht aus, um einen *partiellen* Namen aus dem Quelltext zu *identifizieren*. In Instanzen können *überlagerte* Namen vorkommen. Schon die Namensidentifikation erfordert also eine *Überlagerungsauflösung*. Bei der Berücksichtigung von *generischen* Namen ist sogar eine polymorphe Typanalyse gemäß Algorithmus $\mathcal{W}_o$ erforderlich:

$$\text{Namensidentifikation} \quad = \quad \text{Typanalyse}$$

Die letztendlich plausible Analogie zwischen Namensidentifikation und Typanalyse ist eine im algebraischen Kontext durchaus *neue*, zumindest kaum bewusst verbreitete Erkenntnis: profitieren könnten davon der Entwurf sowie die Implementierung und Verifikation algebraischer Sprachen. Die Verwendung *desselben* Algorithmus für die Analyse von *Namen* und *Ausdrücken* wäre ein exzellentes Beispiel für die immer wieder für korrekte Software propagierte Wiederverwendung (*reuse*).

Mit der Namensidentifikation und einem vorgegebenen Namensraum ist man in der Lage, einen partiellen Namen vollständig aufzulösen bzw. einen Identifizierungsfehler oder eine Mehrdeutigkeit zu diagnostizieren. Die Konstruktion und Erweiterung des Namensraums selbst ist damit aber noch offen.

1.3.4 Namensräume

Ein Namensraum wird syntaktisch durch die partiellen Deklarationen und Importe gebildet – in OPAL an den Schlüsselwörtern TYPE, FUN und IMPORT erkennbar. Doch welchen Namensraum benutzt man, um die partiellen zu vollen Namen zu komplettieren, die letztendlich den Namensraum bilden?

Diese Henne-Ei-Problematik wird in PVS unbefriedigenderweise durch die Beachtung der textuellen Reihenfolge von Deklarationen und Importen gelöst. Ausgehend von einem leeren oder initialen Namensraum, der PVS-*Prelude*, werden die partiellen Namen der ersten Deklaration identifiziert. Nach erfolgreicher Identifizierung wird der Namensraum gemäß der Deklaration erweitert und die nachfolgende Deklaration analog behandelt.

Die sequenzielle Vorgehensweise in PVS entspricht der von ML und wird als *lineare Sichtbarkeit* bezeichnet; in ML werden so die *globale Umgebung* und in PVS die Namensräume von Theorien aufgebaut. Durch die *Überlagerung* in PVS könnten Deklarationen am Anfang einer Theorie im Lichte des Namensraums vom Ende der Theorie als mehrdeutig gelten. Das *Überschreiben* überlagerter Funktionen in ML kann Definitionen von anderen Funktionen hinterlassen, die von früheren und möglicherweise inkompatiblen Versionen abhängen.

Die zu strenge Forderung an die Reihenfolge von Deklarationen hat offensichtliche Nachteile:

- Die erwünschte Lesereihenfolge muss nicht der Analysereihenfolge entsprechen.
- Statt eines großen gemeinsamen Gültigkeitsbereichs haben viele Namen ihren individuellen positionsabhängigen Gültigkeitsbereich.
- Wechselseitige Rekursion (FORWARD in PASCAL) ist problematisch.

Für moderne Programmiersprachen wird ein gewisser Zusatzaufwand der Kontextanalyse in Kauf genommen, um diese Nachteile zu vermeiden. Auch in OPAL darf die Reihenfolge von Deklarationen und Importen sowie Gleichungen und Formeln frei gewählt werden; Deklarationen und Importe dürfen außerdem (mehrfach) wiederholt werden.

Wie erhält man nun aber in OPAL *den* Namensraum, der alle partiellen Deklarationen eindeutig auflöst und genau dadurch etabliert wird? Ist dieser Namensraum selbst eindeutig? Diese Frage wird in Kapitel 8 beantwortet.

1.4 Historie und Motivation

Die vorliegende Arbeit ergab sich aus umfangreichen Programmier-, Lehr- und Forschungsaktivitäten rund um und mit dem seit 1987 an der TU Berlin entwickelten OPAL-Compiler. Hauptziel der Entwicklung von OPAL war der Nachweis, dass mit funktionalen Sprachen produktiv effizienter Code für große Softwaresysteme erzeugt werden kann. Als Beleg dafür wurde der OPAL-Compiler selbst in OPAL programmiert. Die gleiche Zielsetzung wurde etwa zur selben Zeit auch mit anderen funktionalen Sprachen (z.B. HASKELL) intensiv verfolgt. Die historische Wurzel für die algebraische Orientierung von OPAL ist die Münchner CIP-Gruppe [BBD+81], während HASKELL in Glasgow eher aus der nicht weit entfernten Edinburgher ML-Szene entstand.

Die algebraische Orientierung von OPAL versprach zwar eine theoretisch fundierte Semantik, allein für die Namensauflösung in der Signaturanalyse fehlten konkrete Implementierungshinweise. Im Laufe der Zeit entstand so auf evolutionäre Weise eine Signaturanalyse, die immer mehr Quellen korrekt analysierte, aber mittlerweile so komplex und unwartbar ist, dass weitere Fehler kaum mit vertretbarem Aufwand beseitigt werden können. Allein aus pragmatischer Sicht erscheint eine Re-Implementierung fast kontraproduktiv, da die existierende Stabilität des OPAL-Compilers – täglich von Hunderten von Studenten gewürdigt – nicht leicht wieder zu erreichen wäre.

1.4.1 Beweisbarkeit

Im Vordergrund mehrerer Forschungsprojekte mit OPAL, insbesondere zur korrekten Software KORSO [W+92, BJ95], standen *formale Methoden* und die Vision von beweisbar korrekter Software. OPAL wurde (mehrfach) zu einer Spezifikationssprache erweitert und im Projektrahmen eines verifizierten Fachsprachencompilers [EFP94] entstand ein rudimentärer Beweisprüfer, der allerdings nach Projektende nicht weiter verwendet wurde. Bei der im KORSO-Projekt entwickelten Spezifikationssprache SPECTRUM [B+93] standen eine hohe Ausdrucksmächtigkeit (*wide spectrum*) [WDC+95] und eine Softwareentwicklungs*methodik* [PW94] im Vordergrund. (In [BDDG93] wird eine Anwendung beschrieben.)

Die Beschäftigung mit Beweiswerkzeugen führte mich 1994 zum *Prototype Verification System* PVS [OSR93b]. Mit diesem Beweiswerkzeug lassen sich unter anderem klassische Theoreme aus der Schulmathematik zur Arithmetik, Aussagen- und Prädikatenlogik *formal* beweisen. (Die strenge Typisie-

rung verhindert die Formulierung von Russell's Paradoxon: $\{\mathtt{x} \mid \mathtt{x} \notin \mathtt{x}\}$ ist typfalsch.) Das Beweisen ist ziemlich *intuitiv* und *Erkenntnis fördernd*; die interaktiven Beweisschritte entsprechen etwa denen, die man von einem schriftlichen Beweis aus einem Lehrbuch erwarten würde. Offensichtliche Details, die vielfach selbst in sorgfältigen schriftlichen Beweisen unerwähnt bleiben, werden weitgehend vollautomatisch gelöst. Gleichzeitig kann die totale Korrektheit jederzeit anhand der *Beweiskette* überblickt werden. (Die Beweiskette muss lückenlos und zyklusfrei sein: Zirkelschlüsse sind also ausgeschlossen.)

Die Spezifikationsprache PVS ist *funktional-algebraisch* und ähnelt OPAL in Syntax und Semantik auf eine fast erstaunliche Weise, wenn man den unabhängigen Entwurf beider Sprachen berücksichtigt. Die Parametrisierung und Instanziierung mit Typen und Funktionen ist praktisch bis hin zur Schreibweise identisch, die Datentypen haben in beiden Sprachen eine (algebraisch naheliegende) *initiale* Semantik; das Modul- und Annotationskonzept ist fast gleich: in PVS sind *Theorien* und Datentypen [OS97a] die Module. Darüberhinaus unterstützen beide Sprachen *uninstanziierte* Importe und *Reexporte*, problematische Aspekte, die in dieser Arbeit in Kapitel 7 genauer untersucht werden.

Die unterschiedliche Zielsetzung beider Sprachen bewirkte allerdings auch deutliche Unterschiede: in OPAL sind die Funktionen i.A. *partiell*, während PVS mit Hilfe von Subtypen und abhängigen Typen (*dependent types*) die flexible Spezifikation von *totalen* Funktionen fordert; außerdem können in PVS Funktionen nicht *wechselseitig* rekursiv definiert werden – eine für Implementierungssprachen durchaus lästige Beschränkung. Für eine bestenfalls halbautomatische eins-zu-eins Übersetzung zwischen OPAL und PVS wäre Folgendes zu berücksichtigen:

- Wechselseitig rekursive Funktionen aus OPAL müssten für PVS zu einer einzigen rekursiven Funktion zusammengefasst werden; das würde die Lesbarkeit reduzieren.
- Zu rekursiven Funktionen muss allein gemäß der Syntaxregeln von PVS ein *Terminierungsmaß* (MEASURE), eine Beschreibung der Eingabegröße, angegeben werden.
- Für partielle Funktionen muss in PVS der genaue Definitionsbereich (als Subtyp) spezifiziert werden.
- Nicht alle PVS-Funktionen sind ausführbar, insbesondere die (starke) Gleichheit – ein essenzieller Teil der Logik – steht nicht unmittelbar für konstruktive Berechnungen zur Verfügung.

1.4.2 Werkzeugintegration

Die relative Ähnlichkeit von OPAL und PVS führte zu der Überzeugung, dass man mit einer *reinen* funktional-algebraischen Sprache in der Lage sein sollte, ausführbaren Code zu *erzeugen* und zu *verifizieren*. Die automatische (und korrekte) Einbettung von verifiziertem Code in ein lauffähiges Programm existiert praktisch nicht. Eine (z.B. mit PVS) vollständig bewiesene Funktion – allein dieser Umstand ist selten – wird höchstens manuell oder halbautomatisch in eine andere Programmiersprache übertragen, compiliert und ausgeführt; eine andere *Semantik* erhöht dabei zusätzlich die Wahrscheinlichkeit für Transformationsfehler. Der Erfolg des KIDS-Systems [Smi90] beruht sicherlich zum Teil darauf, dass Codererzeugung und Verifikation (bzw. Korrektheit garantierende Transformationen) eng integriert sind und erst dadurch Synergie entsteht.

Die Entwicklung eines integrierten Übersetzungs- und Beweiswerkzeugs, die für OPAL angestrebt wird [Did97], erfordert erhebliches Know-how in Bezug auf Übersetzer- und Beweisertechniken, das zwar prinzipiell vorhanden ist, aber leider mehr oder weniger undokumentiert in verschiedenen Systemen (und Köpfen) verborgen ist.

Um die Brauchbarkeit des Werkzeugs möglichst früh prüfen zu können, bietet sich für Übersetzer die *Selbstapplikation* an. Ein (brauchbarer) Übersetzer für eine (brauchbare) Sprache sollte in derselben Sprache programmiert werden können und sich selbst übersetzen können. (Und in diesem Sinne ist auch OPAL brauchbar.) Das *Bootstrapping*-Problem dafür ist bekannt.

Für das Beweissystem ist nun interessant, wie weit es sich selbst *beweisen* kann. Damit steht die Konsistenz (*soundness*) der eigenen Logik auf dem Prüfstand und könnte maschinell mit einer Genauigkeit untersucht werden, die manuell bei weitem unerreichbar ist. (Vollständigkeit ist nach Gödel nicht gegeben, aber sein Satz zur Unvollständigkeit der Logik wäre formal beweisbar.) Leider sind die meisten Beweissysteme schon älter und nicht in der (reinen) Objektsprache ihrer Logik programmiert: PVS z.B. basiert auf LISP und ist eben keine Implementierungssprache mit einem Übersetzer. Über die Implementierung von Beweissystemen in einer *reinen* funktionalen Programmiersprache wird in [Han99] berichtet.

1.5 Übersicht

In den Kapiteln 2 und 3 werden klassische Algorithmen für die polymorphe Typinferenz und die Überlagerungsauflösung vorgestellt. In Kapitel 4 wird die polymorphe Typinferenz um Überlagerung zum Algorithmus $\mathcal{W}_o$ erweitert. Kapitel 5 erläutert die algebraische Instanziierung im Hinblick auf die Parametrisierung mit Funktionen. Dabei werden die Namensterme formal definiert und das Prüfen von Instanzen auf *Unifikation* zurückgeführt.

In Kapitel 6 wird die Identifizierung partieller Namen erläutert und durch einen zu $\mathcal{W}_o$ analogen Algorithmus $\mathcal{I}$ angegeben. Statt Typen für Funktionen werden mit der Namensidentifikation $\mathcal{I}$ Instanzen für Namen bestimmt. In beiden Fällen wird dabei die Unifikation zur Prüfung der Konsistenz von *Applikationen* verwendet; Typannotationen sind dabei ebenfalls (eine Art von) Applikationen.

Kapitel 7 beschreibt, welche Namensräume durch Importe entstehen. In Kapitel 8 wird die Namensraumanalyse – unabhängig von der textuellen Reihenfolge der Deklarationen – auf die wiederholte Namensidentifikationen zurückgeführt und auf *parametrisierte Strukturen* angewendet. Im Schlusskapitel 9 werden die Ergebnisse und Rückschlüsse für den Sprachentwurf zusammengefasst sowie verwandte und zukünftige Arbeiten angegeben.

Kapitel 2

Polymorphe Typinferenz

In diesem Kapitel werden zwei bekannte Algorithmen zur polymorphen Typinferenz wiederholt. Zunächst werden die benötigten Datenstrukturen eingeführt. Der Abschnitt 2.2 beschreibt die spezifizierenden Typableitungsregeln. Danach folgen die in gewisser Weise dualen Algorithmen $\mathcal{W}$ und $\mathcal{M}$ in den Abschnitten 2.3 bzw. 2.5. Abschnitt 2.4 enthält die von beiden Algorithmen benötigte *Unifikation*.

2.1 Datenstrukturen

Die polymorphe Typinferenz geht von einem *Ausdruck* aus, dem automatisch ein allgemeinster (bzw. *prinzipaler*) *Typ* (Abschnitt 2.1.2) zugeordnet wird. Per Substitution (Abschnitt 2.1.3) kann ein Typ spezialisiert (oder *instanziiert*) werden. Durch Binden (oder *Generalisieren*) von Typvariablen entsteht aus einem Typ ein Typschema (Abschnitt 2.1.4), das innerhalb einer *Umgebung* (Abschnitt 2.1.5) einer Funktion oder Konstanten zugeordnet ist.

2.1.1 Ausdruck

Ein *Ausdruck* ist entweder eine atomare Variable, eine λ-*Abstraktion* oder eine Funktions-*Applikation*. Eine Besonderheit bilden LET-Ausdrücke. Durch LET werden *polymorphe* Funktionen eingeführt, die im Rumpf geeignet *spezialisiert* verwendet werden können. (OPAL und PVS unterstützen nur *monomorphe* LET-Ausdrücke.)

Für *primitive* bzw. in die Sprache *eingebaute* (builtin) Funktionen werden keine expliziten LET-Definitionen angegeben; sie sind Teil der *initialen Umgebung*. Die Grammatik der Ausdrücke von Core-ML [DM82] sieht wie folgt aus:

`expr` ::==	`var`	- *Variable*
	`expr(expr)`	- *Applikation*
	λ`var.expr`	- *Abstraktion*
	LET `var` = `expr` IN `expr`	
	FIX `var.expr`	

Die Schlüsselwörter, runde Klammern „(“, „)“ und die Zeichen „λ“, „$\forall$“, „ . “, „=“ sind *Terminalsymbole*. Die atomaren Variablen `var` sind die Bezeichner für Funktionen oder Konstanten.

Die LET-Variable ist im definierenden Ausdruck (vor IN) unbekannt. Sinnvollerweise sollte die LET-Variable aber im Rumpf (nach IN) appliziert werden.

Für *rekursive* Funktionsdefinitionen existieren die speziellen FIX-Ausdrücke, die ansonsten für die polymorphe Typinferenz keine besondere Rolle spielen. Mit einem Verweis auf den *Fixpunktoperator* λ`f.`(λ`x.f(x(x))`)(λ`x.f(x(x))`), der allerdings nicht typisierbar ist, werden FIX-Ausdrücke häufig (in [Smi91, NN99]) ignoriert. Die FIX-Variable bezeichnet die Funktion, die im Rumpf rekursiv aufgerufen werden kann. Die zumindest für die Terminierung von Rekursionen nötige Fallunterscheidung IF-THEN-ELSE-FI und die Wahrheitswerte sind spezielle λ-Ausdrücke und deswegen nicht Teil der *minimalen* Syntax [Thi94] (S. 265):

`true` = λ`x.`λ`y.x`
`false` = λ`x.`λ`y.y`
IF _ THEN _ ELSE _ FI = λ`z.`λ`x.`λ`y.z(x)(y)`

Die Klammerung mehrerer Applikationen ist linksassoziativ:

$\mathtt{e}_1(\mathtt{e}_2)(\mathtt{e}_3) = (\mathtt{e}_1(\mathtt{e}_2))(\mathtt{e}_3)$

Die operationale Auswertung der Ausdrücke, speziell die nicht-strikte Auswertung der THEN und ELSE-Zweige, ist für die Typanalyse belanglos.

Weiterhin sind *Tupel*, das parallele *nicht-rekursive* LET und LETREC für *wechselseitige* Rekursionen nicht Teil der Syntax von Core-ML. Tupelausdrücke sind spezielle *curried*-Applikationen eines Tupelkonstruktors:

$(\mathtt{e}_1, \ldots, \mathtt{e}_k) = \mathtt{tuple}_k(\mathtt{e}_1) \ldots (\mathtt{e}_k)$

Paare können z.B. durch den Kombinator $\mathtt{tuple}_2 = \lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\mathtt{z}(\mathtt{x})(\mathtt{y})$ mit den Projektionsfunktionen $\mathtt{proj}_{2,1} = \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x}$ und $\mathtt{proj}_{2,2} = \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x}$ (`true` und `false`) kodiert werden ([Thi94] S. 300). Die *musterbasierte* Zerlegung der Tupel ist eine Abkürzung für die explizite Verwendung von Projektionsfunktionen:

$\text{LET } (\mathtt{x}_1, \ldots, \mathtt{x}_k) = \mathtt{t} \text{ IN } \mathtt{e} \Leftrightarrow$
$\text{LET } \mathtt{x}_1 = \mathtt{proj}_{k,1}(\mathtt{t}) \ \ldots \ \mathtt{x}_k = \mathtt{proj}_{k,k}(\mathtt{t}) \text{ IN } \mathtt{e}$

Das parallele nicht-rekursive LET, bei dem keine Variable $\mathtt{x}_i$ *frei* in einem der definierenden Ausdrücke $\mathtt{e}_j$ vorkommt, kann durch Schachtelung simuliert werden. Die Reihenfolge der Gleichungen ist dabei irrelevant:

$\forall 1 \leq \mathtt{i}, \mathtt{j} \leq \mathtt{k}.\ \mathtt{x}_i \notin \mathtt{fv}(\mathtt{e}_j) \Rightarrow \text{LET } \mathtt{x}_1 = \mathtt{e}_1 \ \ldots \ \mathtt{x}_k = \mathtt{e}_k \text{ IN } \mathtt{e}$
$\Leftrightarrow \text{LET } \mathtt{x}_1 = \mathtt{e}_1 \text{ IN LET } \ldots \text{ IN LET } \mathtt{x}_k = \mathtt{e}_k \text{ IN } \mathtt{e}$

Eine Besonderheit ist das parallele LET von OPAL, bei dem genau die Vorkommen der LET-Variablen in den definierenden Ausdrücken eine nicht eindeutige, sequenzielle Reihenfolge festlegen. Falls $\mathtt{x}_i \in \mathtt{fv}(\mathtt{e}_j)$, dann steht die `i`-te Gleichung vor der `j`-ten und Zyklen sind verboten.

Wechselseitige LETREC-Rekursion kann durch eine einfache Rekursion und Tupelausdrücke simuliert werden:

$\text{LETREC } \mathtt{x}_1 = \mathtt{e}_1 \ \ldots \ \mathtt{x}_k = \mathtt{e}_k \text{ IN } \mathtt{e} \Leftrightarrow$
$\text{LET } (\mathtt{x}_1, \ldots, \mathtt{x}_k) = (\text{FIX } \mathtt{t}.\text{LET } (\mathtt{x}_1, \ldots, \mathtt{x}_k) = \mathtt{t} \text{ IN } (\mathtt{e}_1, \ldots, \mathtt{e}_k)) \text{ IN } \mathtt{e}$

Die FIX-Variable `t` steht für das Tupel der wechselseitig rekursiven Funktionen. Allein für die Typanalyse und unabhängig von einer operationalen Semantik muss die FIX-Variable nicht unbedingt eine Funktion `f` sein, wie das explizit in [LY98] durch $\mathtt{expr} ::== \text{FIX } \mathtt{f}.\lambda\mathtt{x}.\mathtt{e}$ verlangt wird.

2.1.2 Typ

Ein *Typ* (bzw. *Typterm*) wird durch *Typkonstruktoren* und *Typvariablen* wie folgt (baumartig) aufgebaut:

`type` ::==	`typeConstr type`*	- *Typkonstruktor mit Argumenten*
	`typeVar`	- *Typvariable (durchnummeriert)*

Das hochgestellte Metasymbol * in Postfix-Notation ist der reguläre Kleene-Stern, der hier die null- oder mehrmalige Wiederholung des *Nonterminals* `type` bezeichnet.

Der elementare Funktionstyp $\rightarrow$ ist ein *zweistelliger* Typkonstruktor und wesentlich für Funktionen höherer Ordnung bzw. Funktionen als „first class citizens“. Der Funktionstyp ist Grundlage für andere bzw. gleichberechtigt zu anderen Typkonstruktoren; die Infix-Notation ist rechtsassoziativ:

$$\mathtt{t_1} \rightarrow \mathtt{t_2} \rightarrow \mathtt{t_3} = \mathtt{t_1} \rightarrow (\mathtt{t_2} \rightarrow \mathtt{t_3})$$

Weitere der endlich vielen Typkonstruktoren sind fast immer die zwei- und mehrstelligen Tupel, mit $\times$ als Infix- bzw. Mixfixsymbol. Die Typen der Tupelkonstruktoren $(\_, \ldots, \_)$ und Projektionsfunktionen lauten damit:

$$\mathtt{tuple_k} : \mathtt{t_1} \rightarrow \ldots \rightarrow \mathtt{t_k} \rightarrow \mathtt{t_1} \times \ldots \times \mathtt{t_k} \quad (\mathtt{n} \geq 2)$$
$$\mathtt{proj_{n,i}} : \mathtt{t_1} \times \ldots \times \mathtt{t_k} \rightarrow \mathtt{t_i} \quad (1 \leq \mathtt{i} \leq \mathtt{n})$$

Die angegebenen Tupel hier unterscheiden sich von den *assoziativen*, d.h. unverschachtelten bzw. „flachgeklopften“ Tupeln aus OPAL, die nicht zum Aufbau der Typkonstruktorterme `type` passen und die auch praktisch eher als Entwurfsfehler wahrgenommen werden, wenn der Unterschied der Tupelarten überhaupt relevant ist. (Eine Instanziierung mit assoziativen Tupeln `seq`$[\mathtt{nat} \times \mathtt{nat}]$ ist in OPAL illegal und muss durch `seq`$[$`pair`$[\mathtt{nat}, \mathtt{nat}]]$ erfolgen.)

Nullstellige Typkonstruktoren sind die Basistypen wie `bool` oder `nat`. Der Typkonstruktor für homogene Listen ist *einstellig*. Die konkreten Notationen für die Typkonstruktorapplikationen unterscheiden sich in ML, HASKELL und OPAL. In ML werden benutzerdefinierte Typkonstruktoren *postfix* notiert, z.B. `Int List`. Eine ungeklammerte Präfix-Notation wird von HASKELL unterstützt. Speziell für die vordefinierten Listen ist die *Outfix*-Notation mit eckigen Klammern `[Int]` verbreitet, die mit der Instanzannotation von OPAL `seq`$[$`nat`$]$ kompatibel sein könnte, wenn statt `seq` ein *unsichtbarer Typkonstruktor* erlaubt wäre.

Die Typvariablen werden mit Hilfe natürlicher Zahlen und einem *Konstruktor* `tvar`$:$ `nat` $\rightarrow$ `type` durch `tvar`$($`n`$)$ repräsentiert. Verbreitet ist auch die Notation mit kleinen griechischen Buchstaben $\alpha, \beta, \ldots$ bzw. in ML $'\mathtt{a}, '\mathtt{b}, \ldots$ als Kurzform für `tvar`$(1), $`tvar`$(2), \ldots$; eine Verwechslung mit λ- oder LET-Variablen, meistens mit $\mathtt{x}, \mathtt{y}, \ldots$ bezeichnet, bzw. eine Vermischung der durchaus gleichartigen Typ- und Ausdrucksebenen sollte damit ausgeschlossen sein.

Die Menge der Typvariablen eines Typterms $\mathtt{tv(t)}$ ist wie folgt musterbasiert rekursiv definiert:

$$\mathtt{tv}(\mathtt{C_k\ t_1 \ldots t_k}) = \mathtt{tv}(\mathtt{t_1}) \cup \ldots \cup \mathtt{tv}(\mathtt{t_k})$$
$$\mathtt{tv}(\mathtt{tvar(n)}) = \{\mathtt{n}\}$$

Dabei ist $\mathtt{C_k}$ ein k-stelliger Typkonstruktor ($\mathtt{k} \geq 0$). Für nullstellige Typkonstruktoren ergibt sich eine leere Menge und für eine einzelne Typvariable ist die Menge einelementig. Mehrfachvorkommen derselben Typvariablen werden nur einmal zur Menge hinzugenommen. Der später bei der Unifikation (Abschnitt 2.4) benötigte *Occurs-Check* wird damit zum Elementtest.

2.1.3 Substitution

Die Typvariablen sind *Blätter* der baumartigen Typterme, die durch Typterme *substituiert* (ersetzt) werden können. Dadurch entsteht wieder ein Typterm vom Typ `type`, der an Stelle der vormaligen Blätter *Teilbäume* aufweist.

Substitutionen werden hier über einen abstrakten Datentyp `subst` modelliert, der einer *endlichen* Abbildung vom Typ $\mathtt{nat} \rightarrow \mathtt{type}$ entspricht, die auch als endliche Liste von Paaren vom Typ $\mathtt{seq[nat} \times \mathtt{type]}$ durch $[\alpha := \mathtt{t_1}, \beta := \mathtt{t_2}, \ldots]$ notiert werden kann.

Der Definitionsbereich einer Substitution $\mathtt{S}$ ist die *endliche* Menge $\mathtt{dom(S)}$ von Typvariablen (als Nummern), die echt verändert werden:

$$\mathtt{dom(S)} = \{\mathtt{n} \in \mathtt{nat} \mid \mathtt{S(n)} \neq \mathtt{tvar(n)}\}$$

Für die leere Substitution ϵ ist der Definitionsbereich leer:

$$\mathtt{S} = \epsilon \ \Leftrightarrow\ \mathtt{dom(S)} = \emptyset$$

Die Anwendung einer Substitution $\mathtt{S}$ auf einen Term $\mathtt{t}$, notiert als Applikation $\mathtt{S(t)}$ (bzw. postfix $\mathtt{t\ S}$), ist wie folgt als paralleler Ersetzungsprozess definiert:

$$\mathtt{S}(\mathtt{C_k\ t_1 \ldots t_k}) = \mathtt{C_k\ S}(\mathtt{t_1}) \ldots \mathtt{S}(\mathtt{t_k})$$
$$\mathtt{S}(\mathtt{tvar(n)}) = \mathtt{S(n)}$$

Dabei gilt $\mathtt{S}(\mathtt{tvar(n)}) = \mathtt{S(n)} = \mathtt{tvar(n)}$ für *fast alle* $\mathtt{n}$. Die Substitution $\mathtt{S}$ lässt also Variablen bzw. ganze Terme invariant, die nicht in $\mathtt{dom(S)}$ vorkommen bzw. keine Variablen aus $\mathtt{dom(S)}$ enthalten:

$$\mathtt{dom(S)} \cap \mathtt{tv(t)} = \emptyset \;\Rightarrow\; \mathtt{S(t)} = \mathtt{t}$$

Die sequenzielle Komposition von Substitutionen entspricht der Funktionskomposition:

$$(\mathtt{S_2} \circ \mathtt{S_1})(\mathtt{t}) = \mathtt{S_2}(\mathtt{S_1}(\mathtt{t})) = \mathtt{t}\ \mathtt{S_1}\ \mathtt{S_2}$$

Eine im Zusammenhang mit dem *Occurs-Check* während der Unifikation wichtige Eigenschaft von Substitutionen ist, dass sie die echte Teiltermbeziehung, notiert durch $\mathtt{t_1} \prec \mathtt{t_2}$, invariant lässt. Diese Eigenschaft heißt in [MW81] *Monotonie*:

$$\mathtt{t_1} \prec \mathtt{t_2} \;\Leftrightarrow\; \forall \mathtt{S}.\ \mathtt{S(t_1)} \prec \mathtt{S(t_2)}$$

Die für die Unifikation relevanten Substitutionen sind *idempotent*:

$$\mathtt{idempotent(S)} \;\Leftrightarrow\; \mathtt{S} \circ \mathtt{S} = \mathtt{S}$$

Bei Anwendung einer idempotenten Substitution S auf einen beliebigen Term t werden die Variablen des Definitionsbereichs dom(S) aus S(t) *entfernt* (herausubstituiert), insbesondere jeder einzusetzende Term, d.h. der Wertebereich der Substitution, enthält keine Variablen des Definitionsbereichs. Die folgenden drei Charakterisierungen für *idempotente* Substitutionen [MW81] sind äquivalent:

$\mathtt{S} \circ \mathtt{S} = \mathtt{S}$ - *(1)*
$\forall \mathtt{t}.\ \mathtt{dom(S)} \cap \mathtt{tv(S(t))} = \emptyset$ - *(2)*
$\forall \mathtt{x} \in \mathtt{dom(S)}.\ \mathtt{dom(S)} \cap \mathtt{tv(S(x))} = \emptyset$ - *(3)*

Durch die Betrachtung der jeweiligen Definitionsbereiche mit Fallunterscheidungen beweist man, dass die sequenzielle Komposition idempotenter Substitutionen idempotent ist.

2.1.4 Typschema

Ein *Typschema* ergibt sich durch *Binden* oder *Generalisieren* einiger (oder aller) Typvariablen eines Typs:

$\mathtt{typeScheme} ::== \forall \mathtt{typeVar}^*.\mathtt{type}$ - *gebundene Typvariablen*

Das Binden auf der äußeren Ebene charakterisiert die *shallow*-Polymorphie.

Die Bindung bewirkt eine disjunkte Zerlegung der Typvariablen in *gebundene* und *freie* Typvariablen:

$$\texttt{bv}(\forall\alpha_1 \ldots \alpha_k.\texttt{t}) = \{\alpha_1, \ldots, \alpha_k\}$$
$$\texttt{fv}(\forall\alpha_1 \ldots \alpha_k.\texttt{t}) = \texttt{tv}(\texttt{t}) \setminus \{\alpha_1, \ldots, \alpha_k\}$$

Ein Typschema mit gebundenen Typvariablen heißt *generisch* oder *polymorph*, ansonsten *monomorph*. Freie bzw. ungebundene Typvariablen heißen auch *Unbekannte*.

Die gebundenen Typvariablen sind *lokale* Bezeichnungen, die umbenannt werden können, solange dadurch nicht verschiedene Typvariablen zusammenfallen; diese Umbenennung nennt sich α-Konversion auf der Typebene und derartig konvertierte Typschemata sind identisch.

Aus einem geeignet α-konvertierten *Typschema* `s` entsteht ein *Typ* `t` durch *Spezialisierung* (oder *Instanziierung*), bei der *alle* gebundenen Typvariablen durch *Typen substituiert* werden und die Bindung wegfällt. Der Typ `t` ist dann eine *Instanz* des Schemas `s`:[1]

$$\texttt{s} \preceq \texttt{t} \;\Leftrightarrow\; \exists \texttt{S}.\; \texttt{dom}(\texttt{S}) = \texttt{bv}(\texttt{s}) \wedge \texttt{S}(\texttt{type}(\texttt{s})) = \texttt{t}$$

Im einfachsten Fall kann ein Typschema mit *frischen* Unbekannten instanziiert werden. Ein bei jeder Instanziierung zu erhöhender globaler Index `n` gibt dabei an, ab welcher Nummer Typvariablen *neu* sind:

$$\texttt{inst}(\forall\alpha_1 \ldots \alpha_k.\texttt{t}, \texttt{n}) = \texttt{t}[\alpha_1 := \texttt{tvar}(\texttt{n}), \ldots, \alpha_k := \texttt{tvar}(\texttt{n} + \texttt{k} - 1)]$$

Die Anwendung einer Substitution `S` auf ein ganzes *Typschema* `s` betrifft immer nur die *freien* Typvariablen:

$$\texttt{dom}(\texttt{S}) \cap \{\alpha_1, \ldots, \alpha_k\} = \emptyset \;\Rightarrow\; \texttt{S}(\forall\alpha_1 \ldots \alpha_k.\texttt{t}) = \forall\alpha_1 \ldots \alpha_k.\texttt{S}(\texttt{t})$$

Typschemata und Typen fallen zusammen, wenn keine Typvariable gebunden wird. Die Typvariablen eines *Typs* `t` sind also immer *frei*, d.h. $\texttt{fv}(\texttt{t}) = \texttt{tv}(\texttt{t})$.

2.1.5 Umgebung

In einer *endlichen Umgebung* `env : var` $\rightarrow$ `typeScheme` wird jeder Variablen aus übergeordneten λ- und LET-Ausdrücken eindeutig ein *Typschema* zuge-

[1] Zwischen der Instanziierungsrelation $\preceq$ und der Teiltermbeziehung $\prec$ besteht kein Zusammenhang

ordnet. Im Typschema für λ-Variablen sind allerdings keine Typvariablen gebunden; nur der Typ von LET-Variablen wird generalisiert.

Darüberhinaus werden für alle eingebauten oder benutzerdefinierten (frei generierten) *Datentypen* automatisch weitere Einträge zur Umgebung hinzugefügt. In erster Linie sind das die *Konstruktoren*, um Datenelemente zu erzeugen, die durch Konstruktorterme repräsentiert werden. Diese Wertkonstruktoren sind polymorph, wenn der zugehörige Typkonstruktor *mehrstellig* ist.

Zur Zerlegung von Daten werden darüberhinaus (in OPAL) Test- und Selektionsfunktionen in die *initiale Umgebung* aufgenommen. Die verbreitete *musterbasierte* Zerlegung (durch *Pattern-Matching* [BGJ89]) kann man als äquivalent zu expliziten Applikationen von Tests (*Diskriminatoren* oder *Recognizers*) und Selektoren (*Accessors* in PVS) betrachten.

Für den polymorphen Datentyp `option` OPAL-ähnlich notiert ergibt sich z.B. folgende initiale Umgebung, die auch als *induzierte Signatur* bezeichnet wird:

TYPE `option`$[\alpha]$ = `nil`	- *optionaler*
`avail`(`cont` : α)	- *Wert*
`nil` : $\forall\alpha.$`seq`$[\alpha]$	- *kein Wert*
`avail` : $\forall\alpha.\alpha \to$ `option`$[\alpha]$	- *Wert einpacken*
`nil?` : $\forall\alpha.$`option`$[\alpha] \to$ `bool`	- *Test auf kein Wert*
`avail?` : $\forall\alpha.$`option`$[\alpha] \to$ `bool`	- *Test auf Wert*
`cont` : $\forall\alpha.$`option`$[\alpha] \to \alpha$	- *Wert auspacken*

Dieser `option`-Datentyp ist isomorph zu den Listen mit maximal einem Element; er ist *monadisch* und repräsentiert *optionale* Ergebnisse, wie sie z.B. von der Unifikation und den Typinferenzalgorithmen berechnet werden. Die Funktionen `avail` und `cont` zum Ein- und Auspacken von Werten muss man sich – wie es in PVS [OSR93a] möglich ist – als *Konversionsfunktionen* vorstellen, die *implizit* appliziert werden.

2.2 Typableitung

Die Behauptung, dass in einer Umgebung (oder im Kontext) `A` der Ausdruck `e` den *Typ* `t` hat, wird als `A` $\vdash$ `e : t` notiert. Diese Relation vom Typ `env` $\times$ `expr` $\times$ `type` $\to$ `bool` wird induktiv und minimal durch folgende Typregeln festgelegt:

$$\frac{\mathtt{A}(\mathtt{x}) \preceq \mathtt{t}}{\mathtt{A} \vdash \mathtt{x} : \mathtt{t}} \quad \text{(VAR)}$$

$$\frac{\mathtt{A} \vdash \mathtt{f} : (\mathtt{t}_1 \to \mathtt{t}_2) \qquad \mathtt{A} \vdash \mathtt{e} : \mathtt{t}_1}{\mathtt{A} \vdash \mathtt{f}(\mathtt{e}) : \mathtt{t}_2} \quad \text{(APPL)}$$

$$\frac{(\mathtt{A} + \mathtt{x} : \mathtt{t}_1) \vdash \mathtt{e} : \mathtt{t}_2}{\mathtt{A} \vdash (\lambda \mathtt{x}.\mathtt{e}) : (\mathtt{t}_1 \to \mathtt{t}_2)} \quad \text{(ABS)}$$

$$\frac{\mathtt{A} \vdash \mathtt{e}_1 : \mathtt{t}_1 \qquad (\mathtt{A} + \mathtt{x}_1 : \mathtt{gen}(\mathtt{A}, \mathtt{t}_1)) \vdash \mathtt{e} : \mathtt{t}}{\mathtt{A} \vdash (\textsc{let}\ \mathtt{x}_1 = \mathtt{e}_1\ \textsc{in}\ \mathtt{e}) : \mathtt{t}} \quad \text{(LET)}$$

$$\frac{(\mathtt{A} + \mathtt{x} : \mathtt{t}) \vdash \mathtt{e} : \mathtt{t}}{\mathtt{A} \vdash (\textsc{fix}\ \mathtt{x}.\mathtt{e}) : \mathtt{t}} \quad \text{(FIX)}$$

$\mathtt{A}(\mathtt{x})$ steht für das Typschema einer durch λ oder LET eingeführten Variablen $\mathtt{x}$ in der Umgebung $\mathtt{A}$. $\mathtt{A}(\mathtt{x}) \preceq \mathtt{t}$ bedeutet, dass der Typ $\mathtt{t}$ (mit freien Typvariablen) aus dem *Typschema* $\mathtt{A}(\mathtt{x})$ durch *Substitution* der *gebundenen* Typvariablen also durch *Instanziierung* entstanden ist. Enthält $\mathtt{A}(\mathtt{x})$ keine gebundenen Variablen, dann ist $\mathtt{A}(\mathtt{x})$ (ohne $\forall$-Quantor) gleich $\mathtt{t}$.

Durch `gen` in der (LET)-Regel werden *Unbekannte* im Typ $\mathtt{t}_1$ *generalisiert*, aber nur solche, die nicht in der Umgebung $\mathtt{A}$ vorkommen! Die *lokal* gebundenen Typvariablen im Typschema für die LET-Variable $\mathtt{x}_1$ dürfen nicht mit den *globalen* Unbekannten aus umfassenden λ-Ausdrücken kollidieren. Nur bei der Instanziierung mit der (VAR)-Regel können Unbekannte aus der Umgebung für gebundene Variablen eingesetzt werden. Eine *ungeeignete* Substitution blockiert allerdings spätere Regelanwendungen, führt also in eine Sackgasse bei der Typableitung.

Frische Unbekannte können entweder per Substitution von gebundenen Variablen in der (VAR)-Regel oder als Annahme $\mathtt{t}_1$ für die λ-Variable $\mathtt{x}$ in der (ABS)-Regel eingeführt werden. Mit $\mathtt{A} + \mathtt{x} : \mathtt{t}_1$ wird $\mathtt{t}_1$ als triviales Typschema ohne gebundene Variablen für $\mathtt{x}$ zur Umgebung $\mathtt{A}$ hinzugefügt. Analoges gilt für die (FIX)-Regel.

Durch $\mathtt{A} + \mathtt{x} : \mathtt{s}$ mit Typ $\mathtt{env} \times \mathtt{var} \times \mathtt{typeScheme} \to \mathtt{env}$ wird die Umgebung *erweitert*. Wenn $\mathtt{x}$ allerdings schon in $\mathtt{A}$ enthalten ist, wird das Typschema für $\mathtt{x}$ mit $\mathtt{s}$ *überschrieben*, d.h. es gilt immer: $(\mathtt{A} + \mathtt{x} : \mathtt{s})(\mathtt{x}) = \mathtt{s}$. Die Variablen aus inneren λ- oder LET-Ausdrücken *verschatten* also solche aus äußeren.

Mit Hilfe dieser Ableitungsregeln kann man nun *spezifizieren*, wann ein Ausdruck $\mathtt{e}$ typkorrekt ist, nämlich dann, wenn man für eine vorgegebene Umgebung $\mathtt{A}$ und einen zu ermittelnden Typ $\mathtt{t}$ die Beziehung $\mathtt{A} \vdash \mathtt{e} : \mathtt{t}$ ableiten kann.

Für den Ausdruck $\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}(\mathtt{f}(\mathtt{x}))$ [Smi91] lässt sich die Korrektheit der Typisierung $(\gamma \to \gamma) \to (\gamma \to \gamma)$ mit $[\mathtt{f} := \gamma \to \gamma, \mathtt{x} := \gamma]$ als *geeignete* Substitutionen wie folgt ableiten:

$\{\} \vdash (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}(\mathtt{f}(\mathtt{x}))) : (\gamma \to \gamma) \to (\gamma \to \gamma)$ - *(ABS)*
$\{\mathtt{f} : \gamma \to \gamma\} \vdash (\lambda\mathtt{x}.\mathtt{f}(\mathtt{f}(\mathtt{x}))) : \gamma \to \gamma$ - *(ABS)*
$\{\mathtt{f} : \gamma \to \gamma, \mathtt{x} : \gamma\} \vdash \mathtt{f}(\mathtt{f}(\mathtt{x})) : \gamma$ - *(APPL)*
$\{\mathtt{f} : \gamma \to \gamma, \mathtt{x} : \gamma\} \vdash \mathtt{f}(\mathtt{x}) : \gamma$ - *(APPL)*
$\{\mathtt{f} : \gamma \to \gamma, \mathtt{x} : \gamma\} \vdash \mathtt{f} : \gamma \to \gamma$ - *(VAR)*
$\{\mathtt{f} : \gamma \to \gamma, \mathtt{x} : \gamma\} \vdash \mathtt{x} : \gamma$ - *(VAR)*

Allein durch die Ableitungsregeln ist der Typ für einen Ausdruck noch nicht festgelegt. Derselbe Ableitungsprozess würde z.B. für jeden spezielleren Typ genauso funktionieren. Die Typableitungsrelation ist *abgeschlossen bezüglich Substitution*, d.h. für alle $\mathtt{A}, \mathtt{e}, \mathtt{t}, \mathtt{S}$ gilt:

$$\mathtt{A} \vdash \mathtt{e} : \mathtt{t} \;\Rightarrow\; \mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{S}(\mathtt{t})$$

Eine weitere wichtige Eigenschaft ist, dass immer ein *allgemeinster* (prinzipaler) Typ $\mathtt{t}_0$ zu $\mathtt{e}$ mit $\mathtt{A} \vdash \mathtt{e} : \mathtt{t}_0$ existiert, der bis auf Umbenennung von Unbekannten eindeutig ist. Alle anderen Typen $\mathtt{t}$ mit $\mathtt{A} \vdash \mathtt{e} : \mathtt{t}$ sind dann *Spezialisierungen* von $\mathtt{t}_0$, d.h. es gibt eine Substitution $\mathtt{R}$ mit $\mathtt{t} = \mathtt{R}(\mathtt{t}_0)$:

$$\mathtt{A} \vdash \mathtt{e} : \mathtt{t} \;\Rightarrow\; \exists \mathtt{t}_0\ \mathtt{R}.\ \mathtt{A} \vdash \mathtt{e} : \mathtt{t}_0 \;\wedge\; \mathtt{t} = \mathtt{R}(\mathtt{t}_0)$$

2.3 Algorithmus $\mathcal{W}$

Die *operationale* Berechnung des *prinzipalen* Typs ist mit dem Algorithmus $\mathcal{W}$ möglich. Die Ein- und Ausgabe vom Typ `nat` ist rein technischer Natur und dient der Verwaltung eines globalen Zählers für *frische* Typvariablen.

FUN $\mathtt{W}$: $\mathtt{env} \times \mathtt{expr} \times \mathtt{nat} \to \mathtt{option}[\mathtt{type} \times \mathtt{subst} \times \mathtt{nat}]$

Der Algorithmus liefert im Erfolgsfall aus einer Umgebung $\mathtt{A}$ und einem Ausdruck $\mathtt{e}$ ein Tripel: den prinzipalen Typ $\mathtt{t}$, eine Substitution $\mathtt{S}$ und eine Nummer. Die Substitution $\mathtt{S}$ enthält dabei *Spezialisierungen* für Unbekannte in der Umgebung $\mathtt{A}$, die z.B. zwischenzeitlich für Typen von λ-Variablen hinzukommen. Falls der Eingabeausdruck typfalsch ist, lautet die Ausgabe $\mathtt{nil} : \mathtt{option}$.

Konsistenz

$$\mathtt{W}(\mathtt{A}, \mathtt{e}, \_) = (\mathtt{t}, \mathtt{S}, \_) \;\Rightarrow\; \mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{t}$$

Die formale Teilspezifikation für typkorrekte Ausdrücke `e` wird als *Konsistenz* (soundness) bezeichnet. Aus einer erfolgreichen Berechnung ergibt sich die Ableitbarkeit gemäß der Typisierungsregeln. Für die Typinferenz eines *geschlossenen* Ausdrucks ist die initiale Umgebung leer bzw. enthält vordefinierte Funktionen mit Typen *ohne Unbekannte*. Am Ende der Typinferenz gilt also $\mathtt{S}(\mathtt{A}) = \mathtt{A}$ und damit $\mathtt{A} \vdash \mathtt{e} : \mathtt{t}$. Für die Berechnungsergebnisse `S` und `t` gilt außerdem $\mathtt{S}(\mathtt{t}) = \mathtt{t}$, der Ausgabetyp `t` liegt minimal spezialisiert vor und die Substitution `S` ist *idempotent*.

Vollständigkeit

$$\mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{t} \;\Rightarrow\; \exists \mathtt{t}_0\ \mathtt{R}.\ \mathtt{W}(\mathtt{A}, \mathtt{e}, \_) = (\mathtt{t}_0, \mathtt{S}, \_) \;\wedge\; \mathtt{t} = \mathtt{R}(\mathtt{t}_0)$$

Die Typinferenz ist *vollständig*, d.h. falls ein Ausdruck `e` gemäß der Ableitungsregeln typkorrekt mit Typ `t` ist, dann liefert der Algorithmus einen *prinzipalen* Typ $\mathtt{t}_0$, der bis auf Umbenennungen eindeutig und *allgemeiner* als `t` ist; d.h. $\mathtt{t}_0$ kann durch eine Substitution `R` zu `t` spezialisiert werden.

Der bisher ignorierte globale Zähler `n` für *frische* Typvariablen muss *vor* der Berechnung immer größer als alle Unbekannten in der Umgebung `A` sein. *Nach* der Berechnung ist er dann außerdem größer als alle Unbekannten im Ergebnis `t` und `S` [NN99]. Weitere Erläuterungen zum Algorithmus werden anschließend gegeben:

$\mathtt{W}(\mathtt{A},\mathtt{x},\mathtt{n}) =$
    IF $\mathtt{x} \in \mathtt{A}$
        THEN $(\mathtt{inst}(\mathtt{n}, \mathtt{A}(\mathtt{x})), \epsilon, \mathtt{n} + \#(\mathtt{bv}(\mathtt{A}(\mathtt{x}))))$
        ELSE `nil` - *unbekannte Variable*
    FI

$\mathtt{W}(\mathtt{A},\mathtt{f}(\mathtt{e}),\mathtt{n}) =$
    $(\mathtt{t}, \mathtt{S}_1, \mathtt{m}) := \mathtt{W}(\mathtt{A}, \mathtt{f}, \mathtt{n});$
    $(\mathtt{t}_1, \mathtt{S}_2, \mathtt{l}) := \mathtt{W}(\mathtt{S}_1(\mathtt{A}), \mathtt{e}, \mathtt{m});$
    $\mathtt{S}_3 := \mathtt{unify}(\mathtt{S}_2(\mathtt{t}), \mathtt{t}_1 \to \mathtt{tvar}(\mathtt{l}));$
    $(\mathtt{S}_3(\mathtt{l}), \mathtt{S}_3 \circ \mathtt{S}_2 \circ \mathtt{S}_1, \mathtt{l} + 1)$

$\mathtt{W}(\mathtt{A},\lambda\mathtt{x.e},\mathtt{n}) =$
$\quad(\mathtt{t},\mathtt{S},\mathtt{m}) := \mathtt{W}(\mathtt{A}+\mathtt{x}:\mathtt{tvar}(\mathtt{n}),\mathtt{e},\mathtt{n}+1);$
$\quad(\mathtt{S}(\mathtt{n})\rightarrow\mathtt{t},\mathtt{S},\mathtt{m})$

$\mathtt{W}(\mathtt{A},\text{LET } \mathtt{x}_1=\mathtt{e}_1\text{IN } \mathtt{e},\mathtt{n}) =$
$\quad(\mathtt{t}_1,\mathtt{S}_1,\mathtt{m}) := \mathtt{W}(\mathtt{A},\mathtt{e}_1,\mathtt{n});$
$\quad(\mathtt{t},\mathtt{S}_2,\mathtt{l}) := \mathtt{W}(\mathtt{S}_1(\mathtt{A})+\mathtt{x}_1:\mathtt{gen}(\mathtt{S}_1(\mathtt{A}),\mathtt{t}_1),\mathtt{e},\mathtt{m});$
$\quad(\mathtt{t},\mathtt{S}_2\circ\mathtt{S}_1,\mathtt{l})$

$\mathtt{W}(\mathtt{A},\text{FIX } \mathtt{x.e},\mathtt{n}) =$
$\quad(\mathtt{t},\mathtt{S}_1,\mathtt{m}) := \mathtt{W}(\mathtt{A}+\mathtt{x}:\mathtt{tvar}(\mathtt{n}),\mathtt{e},\mathtt{n}+1);$
$\quad\mathtt{S}_2 := \mathtt{unify}(\mathtt{S}_1(\mathtt{n}),\mathtt{t});$
$\quad(\mathtt{S}_2(\mathtt{t}),\mathtt{S}_2\circ\mathtt{S}_1,\mathtt{m})$

Die Notation ist nur pseudo-formal bezüglich Pattern-Matching, Mixfix-Notation und Typkonversion. Eine *erfolgreiche* Berechnung liefert ein Tripel (vom Typ `option`), das in der nachfolgenden Berechnung benutzt werden kann; die scheinbar imperative Schreibweise [NN99] $\mathtt{V} := \mathtt{S};\mathtt{E}$ muss rein funktional als monadische Komposition $\mathtt{S};(\lambda\mathtt{V}.\mathtt{E})$ des `option`-Datentyps[2] gelesen werden. Falls der erste Ausdruck `S` schon `nil` liefert, wird `E` nicht ausgewertet und das Gesamtergebnis ist `nil`, also typfalsch.

Im (VAR)-Fall muss sich die Variable `x` in der Umgebung `A` befinden, also durch einen übergeordneten λ- oder LET-Ausdruck eingeführt worden sein. Das Typschema $\mathtt{A}(\mathtt{x})$ zu dieser Variablen wird anschließend mit frischen Unbekannten durch `inst` instanziiert und der Zähler gemäß der Anzahl der gebundenen Variablen $\#(\mathtt{bv}(\mathtt{A}(\mathtt{x})))$ erhöht; die Ergebnissubstitution ist in diesem Fall leer (ϵ).

Im (APPL)-Fall wird zunächst der Typ für die Funktion und anschließend der Typ für das Argument berechnet. Die Reihenfolge ist im Prinzip beliebig, entscheidend ist aber, dass die Ergebnissubstitutionen aus beiden Berechnungen kombiniert werden. Daher fließt die Substitution (und der potenziell erhöhte Zähler) aus der ersten Berechnung direkt in die nachfolgende ein.

Mit Hilfe der *Unifikation*, die ein Ergebnis vom Typ $\mathtt{option}[\mathtt{subst}]$ liefert, wird erstens zunächst überprüft, ob der Typ der Funktion wirklich ein Funktionstyp ist und zweitens, sofern dem so ist, ob und unter welcher Substitution der Argumenttyp der Funktion zum aktuellen Typ des Arguments passt.

[2] Der Typ der „;“-Komposition ist: $\mathtt{option}[\alpha]\times(\alpha\rightarrow\mathtt{option}[\beta])\rightarrow\mathtt{option}[\beta]$

Der Ergebnistyp kann schließlich aus der Substitution für die bei der Funktionstypkonstruktion eingeführte Hilfsvariable $\texttt{tvar}(\texttt{l})$ extrahiert werden. Zuletzt müssen alle nötigen Substitutionen sequenziell in der gefundenen Reihenfolge komponiert werden.

Im (ABS)-Fall wird für die λ-Variable $\texttt{x}$ eine frische Unbekannte $\texttt{tvar}(\texttt{n})$ zur Umgebung hinzugefügt, die bei der Analyse des Rumpfes so weit wie nötig spezialisiert wird und dann der Ergebnissubstitution durch $\texttt{S}(\texttt{n})$ entnommen werden kann. Der Ergebnistyp der λ-Abstraktion ist gleich zum Typ $\texttt{t}$ des Rumpfes.

Im (LET)-Fall passiert genau das, was durch die entsprechende Typregel vorgegeben ist. Lediglich die Substitutionen und Zähler müssen korrekt verwaltet bzw. propagiert werden. Durch `gen` werden die Typvariablen gebunden, die nicht durch den umgebenden Kontext $\texttt{S}_1(\texttt{A})$ festgelegt sind.

Der Algorithmus terminiert garantiert, da in jedem Zweig rekursiv immer echt kleinere Teilausdrücke analysiert werden. Kein grundsätzlich neues Problem birgt der (FIX)-Fall. Außer bei einem unbekannten Symbol im (VAR)-Fall werden Typfehler im Wesentlichen durch *Unifikation* (Abschnitt 2.4) entdeckt.

2.3.1 Ausführliches Beispiel

Die Typinferenz für den Ausdruck $\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f}(\texttt{f}(\texttt{x}))$ [Smi91] verläuft wie folgt:

$$
\begin{array}{ll}
\texttt{W}(\{\}, \lambda\texttt{f}.\lambda\texttt{x}.\texttt{f}(\texttt{f}(\texttt{x})), 1) & 1 \\
\quad \texttt{W}(\{\texttt{f} : \alpha\}, \lambda\texttt{x}.\texttt{f}(\texttt{f}(\texttt{x})), 2) & 2 \\
\quad\quad \texttt{W}(\{\texttt{f} : \alpha, \texttt{x} : \beta\}, \texttt{f}(\texttt{f}(\texttt{x})), 3) & 3 \\
\quad\quad\quad (\alpha, \epsilon, 3) = \texttt{W}(\{\texttt{f} : \alpha, \texttt{x} : \beta\}, \texttt{f}, 3) & 4 \\
\quad\quad\quad \texttt{W}(\{\texttt{f} : \alpha, \texttt{x} : \beta\}, \texttt{f}(\texttt{x}), 3) & 5 \\
\quad\quad\quad\quad (\alpha, \epsilon, 3) = \texttt{W}(\{\texttt{f} : \alpha, \texttt{x} : \beta\}, \texttt{f}, 3) & 6 \\
\quad\quad\quad\quad (\beta, \epsilon, 3) = \texttt{W}(\{\texttt{f} : \alpha, \texttt{x} : \beta\}, \texttt{x}, 3) & 7 \\
\quad\quad\quad\quad [\alpha := \beta \to \gamma] = \texttt{unify}(\alpha, \beta \to \gamma) & 8 \\
\quad\quad\quad (\gamma, [\alpha := \beta \to \gamma], 4) & 9 \\
\quad\quad\quad [\beta, \delta := \gamma] = \texttt{unify}(\beta \to \gamma, \gamma \to \delta) & 10 \\
\quad\quad (\gamma, [\beta, \delta := \gamma, \alpha := \gamma \to \gamma], 5) & 11 \\
\quad (\gamma \to \gamma, [\beta, \delta := \gamma, \alpha := \gamma \to \gamma], 5) & 12 \\
((\gamma \to \gamma) \to (\gamma \to \gamma), [\beta, \delta := \gamma, \alpha := \gamma \to \gamma], 5) & 13
\end{array}
$$

In der obigen Präsentation wurden die Zählerwerte $1, 2, \ldots$ mit den Typva-

riablen $\alpha, \beta, \ldots$ identifiziert, um die Lesbarkeit der Typen zu erleichtern. Die Ergebnisse der Aufrufe von `W` in den Zeilen 1, 2, 3 und 5 stehen passend eingerückt in den Zeilen 13, 12, 11 und 9.

Für die λ-Variablen `f` und `x` wählt man neue Typvariablen α und β als Unbekannte (Zeile 2 und 3). Aus der Applikation `f(x)` (Zeile 5) ergibt sich, dass α ein Funktionstyp sein muss mit Argumenttyp β; eine neue Typvariable γ wird der zugehörige Ergebnistyp. Bei der äußeren Applikation von `f` mit Typ $\beta \to \gamma$ auf das Argument `f(x)` vom Typ γ liefert die Unifikation $\beta = \gamma$ (Zeile 10). (Jetzt wäre eine neue Typvariable δ für den schon bekannten Ergebnistyp γ in Zeile 11 überflüssig.) Für die Abstraktion λ`x.f(f(x))` (Zeile 2) ergibt sich aus der Umgebung zu `x` *und der Substitution* die Gleichheit von Argument- und Ergebnistyp (Zeile 12), also hier der Typ $\gamma \to \gamma$. Derselbe Typ entsteht auch für `f` aus der Umgebung (Zeile 1), da durch die Komposition der Substitutionen α zunächst zu $\beta \to \gamma$ und dann zu $\gamma \to \gamma$ wird. Der Gesamttyp (Zeile 13) ist damit $(\gamma \to \gamma) \to (\gamma \to \gamma)$, der schon in Abschnitt 2.2 korrekt abgeleitet wurde. Nun weiß man auch, dass dieser Typ *prinzipal* ist!

Der λ-Ausdruck λ`x.x(x)` ist typfalsch. Die Unifikation bei der Applikation `x(x)`, nämlich `unify`$(\alpha, \alpha \to \beta)$, scheitert am so genannten *Occurs-Check*; die (APPL)-Regel ist für keine Umgebung mit einem *festen* (monomorphen) Typ für `x` anwendbar. Eine analoger Typfehler ergibt sich auch für den Fixpunktoperator (oder Y-Kombinator) λ`f.`$(\lambda$`x.f(x(x)))`$(\lambda$`x.f(x(x)))`; und dies ist ein Grund für die speziellen FIX-Ausdrücke.

2.3.2 LET-Ausdrücke

Erst durch LET-Ausdrücke entstehen *polymorphe* Typschemata in der Umgebung, mit der eine Selbstapplikation wie `x(x)` typisierbar sein kann:

LET `x` $= \lambda$`y.y` IN `x(x)`

Der Typ für die Identitätsfunktion λ`y.y` ist $\alpha \to \alpha$. Nach der Generalisierung für LET wird `x`$: \forall\alpha.\alpha \to \alpha$ zur Umgebung hinzugefügt. In dieser Umgebung kann mit der (VAR)-Regel sowohl `x`$: \beta \to \beta$ als auch `x`$: (\beta \to \beta) \to (\beta \to \beta)$ abgeleitet werden, d.h. der Typ für `x(x)` und für den gesamten LET-Ausdruck ergibt sich zu $\beta \to \beta$. Dabei ist die Typvariable β *frei* d.h. *ungebunden* – im Gegensatz zum *gebundenen* α im Typschema zu `x`. Ein anderes Beispiel:

λ`x.`LET `y` $=$ `x` IN `y`

Durch den äußeren λ-Ausdruck wird $\mathtt{x}: \alpha$ in die Umgebung eingetragen und deswegen beim Typschema für $\mathtt{y}$ die Unbekannte α *nicht* generalisiert. Eine Applikation $\mathtt{y}(\mathtt{y})$ wäre wieder typfalsch; so ergibt sich als Gesamttyp für den λ-Ausdruck $\alpha \to \alpha$ und α ist nach wie vor *frei*.

Analog zur operationalen Auswertung zur Laufzeit kann man sich auch die Typanalyse von LET-Ausdrücken zur Übersetzungszeit vorstellen. Alle Vorkommen der LET-Variablen werden durch den definierenden Ausdruck *substituiert*. (Dieses ist eine Substitution für Ausdrücke und nicht wie bei der Typinferenz für Typterme.) Durch geeignete α-Konversionen von LET- oder λ-*gebundenen* Variablen ($\mathtt{bv}$) muss die Verschattung *freier* Variablen ($\mathtt{fv}$) vermieden werden:

$$\mathtt{fv}(\mathtt{e}_1) \cap \mathtt{bv}(\mathtt{e}) = \emptyset \wedge \mathtt{x}_1 \notin \mathtt{bv}(\mathtt{e}) \wedge \text{LET } \mathtt{x}_1 = \mathtt{e}_1 \text{ IN } \mathtt{e} \;\Rightarrow\; \mathtt{e}[\mathtt{x}_1 := \mathtt{e}_1]$$

Die substituierten Ausdrücke werden nun für jedes Vorkommen der LET-Variablen getrennt und in verschiedenen Kontexten analysiert. Aus dem Ausdruck LET $\mathtt{x} = \lambda\mathtt{y}.\mathtt{y}$ IN $\mathtt{x}(\mathtt{x})$ wird so semantisch und typgleich $(\lambda\mathtt{y}.\mathtt{y})(\lambda\mathtt{y}.\mathtt{y})$ und aus $\lambda\mathtt{x}.$ LET $\mathtt{y} = \mathtt{x}$ IN $\mathtt{y}$ wird $\lambda\mathtt{x}.\mathtt{x}$. Ein Typfehler könnte allerdings dann unentdeckt bleiben, wenn die LET-gebundene Variable gar nicht im Rumpf vorkommt, also für unbenutzten (*dead*) Code steht.

Das *polymorphe* LET darf nicht mit dem *monomorphen* LET verwechselt werden. Ein monomorphes LET $\mathtt{x}_1 = \mathtt{e}_1$ IN $\mathtt{e}$ wäre gleichbedeutend mit $(\lambda\mathtt{x}_1.\mathtt{e})(\mathtt{e}_1)$ und gar nicht mehr Teil einer *minimalen* Syntax[3]. Das (oberste) Beispiel LET $\mathtt{x} = \lambda\mathtt{y}.\mathtt{y}$ IN $\mathtt{x}(\mathtt{x})$ ist polymorph typkorrekt, würde aber *monomorph* zu $(\lambda\mathtt{x}.\mathtt{x}(\mathtt{x}))(\lambda\mathtt{y}.\mathtt{y})$ transformiert und als typfalsch zurückgewiesen.

2.3.3 Komplexität

Die Typinferenz für die polymorphen LET-Ausdrücke ist als NP-Problem bekannt, d.h. das Laufzeitverhalten ist exponentiell, wie das folgende Beispiel aus [HM94, Sch95] verdeutlicht:

$$\begin{array}{l}\text{LET } \mathtt{x} = \ldots \text{ IN}\\ \quad \text{LET } \mathtt{x}_1 = \lambda\mathtt{y}.\mathtt{x}(\mathtt{x}(\mathtt{y})) \text{ IN}\\ \qquad \text{LET } \mathtt{x}_2 = \lambda\mathtt{y}.\mathtt{x}_1(\mathtt{x}_1(\mathtt{y})) \text{ IN}\\ \qquad\quad \ldots\\ \qquad\qquad \text{LET } \mathtt{x}_n = \lambda\mathtt{y}.\mathtt{x}_{n-1}(\mathtt{x}_{n-1}(\mathtt{y})) \text{ IN } \mathtt{x}_n\end{array}$$

[3]Core-ML ist deswegen nicht redundanzfrei; die Äquivalenz monomorpher LET- mit λ-Ausdrücken ist eigentlich eine Beweisverpflichtung.

In $\mathtt{x_1}$ kommt $\mathtt{x}$ zweimal vor. Die textuelle Substitution in $\mathtt{x_2}$ und β-Reduktion liefert $\mathtt{x_2} = \lambda\mathtt{y}.(\lambda\mathtt{y}.\mathtt{x}(\mathtt{x}(\mathtt{y})))(\mathtt{x}(\mathtt{x}(\mathtt{y}))) = \lambda\mathtt{y}.\mathtt{x}(\mathtt{x}(\mathtt{x}(\mathtt{x}(\mathtt{y}))))$, also vier Vorkommen von $\mathtt{x}$. Weiter so verdoppelt kommt $\mathtt{x}$ in $\mathtt{x_n}$ dann $\mathtt{2^n}$ mal vor:

$$\mathtt{x_n} = \lambda\mathtt{y}.\mathtt{x}(\ldots(\mathtt{x}(\mathtt{x}(\mathtt{y})))\ldots) = \lambda\mathtt{y}.\mathtt{x}^{\mathtt{2^n}}(\mathtt{y})$$

In [Sch95] ergibt sich $\forall\alpha\beta.\alpha \to (\alpha \to \alpha \to \beta) \to \beta$ als Typ für $\mathtt{x}$:

LET $\mathtt{x} = \lambda\mathtt{y}.\lambda\mathtt{f}.\mathtt{f}(\mathtt{y})(\mathtt{y})$ IN
  LET $\mathtt{x_1} = \ldots$ IN
    $\ldots$
      LET $\mathtt{x_n} = \ldots$ IN $\mathtt{x_n}$

Die Typisierung von $\mathtt{x}^{\mathtt{i}}(\mathtt{y})$ enthält nun $\mathtt{i}+1$ verschiedene Typvariablen; für $\mathtt{x_n}$ sind das exponentiell $\mathtt{2^n}+1$ viele:

$$\begin{aligned}
&\mathtt{x}(\mathtt{y}) : (\alpha_1 \to \alpha_1 \to \beta_1) \to \beta_1\\
&\mathtt{x}(\mathtt{x}(\mathtt{y})) : (\alpha_2 \to \alpha_2 \to \beta_2) \to \beta_2[\alpha_2 := \alpha_1 \to \alpha_1 \to \beta_1]\\
&\ldots\\
&\mathtt{x}^{\mathtt{i}}(\mathtt{y}) : (\alpha_{\mathtt{i}} \to \alpha_{\mathtt{i}} \to \beta_{\mathtt{i}}) \to \beta_{\mathtt{i}}[\alpha_{\mathtt{i}} := \alpha_{\mathtt{i}-1} \to \alpha_{\mathtt{i}-1} \to \beta_{\mathtt{i}-1}]\ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad \ldots[\alpha_2 := \alpha_1 \to \alpha_1 \to \beta_1]
\end{aligned}$$

Die sequenzielle Ausführung von Substitutionen verdoppelt außerdem in jedem Schritt die Typgröße gemessen als Anzahl der Vorkommen von Unbekannten:

Ausdruck	Typgröße
$\mathtt{x}(\mathtt{y})$	4
$\mathtt{x}(\mathtt{x}(\mathtt{y}))$	$2 + 3 * 2 = 8$
$\ldots$	
$\mathtt{x}^{\mathtt{i+1}}(\mathtt{y})$	$2^{\mathtt{i}} + 3 * 2^{\mathtt{i}} = 2^{\mathtt{i+2}}$
$\ldots$	
$\mathtt{x}^{\mathtt{2^n}}(\mathtt{y})$	$2^{2^{\mathtt{n}}+1}$

Der expandierte Typ zu $\mathtt{x}^{\mathtt{i}}(\mathtt{y})$ enthält zur einen Hälfte $\mathtt{2^i}$ β-Unbekannte $(\beta_1, \ldots, \beta_{\mathtt{i}})$ und zur anderen Hälfte $\mathtt{2^i}$ mal die Unbekannte α_1. Beim Übergang von $\mathtt{i}$ nach $\mathtt{i}+1$ mit einer Indexverschiebung und der Betrachtung der letzten Substitution $[\alpha_2 := \alpha_1 \to \alpha_1 \to \beta_1]$ bleiben $\mathtt{2^i}$ β-Unbekannte $(\beta_2, \ldots, \beta_{\mathtt{i+1}})$ unverändert und jedes α_2 wird verdreifacht. Der Typ zu $\mathtt{x}^{\mathtt{i+1}}(\mathtt{y})$ enthält damit $\mathtt{2} * \mathtt{2^i} = \mathtt{2^{i+1}}$ mal die Unbekannte α_1 und ebensoviele β-Unbekannte (mit $\mathtt{2^i}$ Vorkommen von β_1).

Die Typgröße der Applikation $\mathtt{x_3}(\lambda \mathtt{z.z})$ verdoppelt durch den Argumenttyp $\alpha \to \alpha$ ist $2{*}2^9 = 1024$ und enthält neun verschiedene Unbekannte $'\mathtt{a}, \ldots, '\mathtt{i}$. Mit Klammern, Pfeilen und Leerzeichen füllt dieser Typ gut zwei Seiten in [Sch95].

Die *hyperexponentielle* Typgröße $\mathtt{O}(2^{2^n})$ könnte man zwar vermeiden und auf $\mathtt{O}(2^n)$ reduzieren, indem man die sequenziellen Substitutionen nicht ausführt; praktisch ist eine solche Optimierung aber bedeutungslos und – so weit mir bekannt – nirgendwo implementiert. Theoretisch ist die Typinferenz exponentiell zeitaufwendig, d.h. in der Klasse $\mathtt{DTIME}[2^n]$ [KHM94], was einige Jahre (bis 1989 von Harry G. Mairson [Sch95]) unentdeckt blieb.

2.4 Unifikation

Falls zwei Typen $\mathtt{t_1}, \mathtt{t_2}$ unifizierbar sind, dann liefert die Unifikation eine Substitution $\mathtt{U}$, die beide Typen gleich $\mathtt{t}$ macht, und ansonsten $\mathtt{nil}$:

$$\text{FUN } \mathtt{unify} : \mathtt{type} \times \mathtt{type} \to \mathtt{option}[\mathtt{subst}]$$

$$\mathtt{unify}(\mathtt{t_1}, \mathtt{t_2}) = \mathtt{U} \quad \Rightarrow \quad \exists \mathtt{t}.\ \mathtt{U}(\mathtt{t_1}) = \mathtt{U}(\mathtt{t_2}) = \mathtt{t} \ \vee \ \mathtt{U} = \mathtt{nil}$$

Das Ergebnis $\mathtt{U}$ ist der *allgemeinste* Unifier, d.h. für alle anderen Substitutionen $\mathtt{S}$ mit $\mathtt{S}(\mathtt{t_1}) = \mathtt{S}(\mathtt{t_2})$ gilt, dass es eine Substitution $\mathtt{R}$ gibt mit $\mathtt{R} \circ \mathtt{U} = \mathtt{S}$, d.h. $\mathtt{S}$ ist spezieller als $\mathtt{U}$. Das *Unifikationstheorem* lautet:

$$\mathtt{S}(\mathtt{t_1}) = \mathtt{S}(\mathtt{t_2}) \quad \Leftrightarrow \quad \exists \mathtt{U}\ \mathtt{R}.\ \mathtt{unify}(\mathtt{t_1}, \mathtt{t_2}) = \mathtt{U} \ \wedge \ (\mathtt{R} \circ \mathtt{U} = \mathtt{S})$$

$\mathtt{U}$ ist eindeutig bis auf Umbenennungen und *idempotent*, insbesondere gilt also $\mathtt{U}(\mathtt{t}) = \mathtt{t}$; darüberhinaus enthält $\mathtt{t}$ keine Typvariablen, die nicht schon in $\mathtt{t_1}$ oder $\mathtt{t_2}$ vorkommen:

$$\mathtt{tv}(\mathtt{t_1}) \cup \mathtt{tv}(\mathtt{t_2}) = \mathtt{tv}(\mathtt{t}) \cup \mathtt{dom}(\mathtt{U}) \quad \wedge \quad \mathtt{tv}(\mathtt{t}) \cap \mathtt{dom}(\mathtt{U}) = \emptyset$$

Ohne Typvariablen in $\mathtt{t_1}$ und $\mathtt{t_2}$ ist die Unifikation ein reiner Gleichheitstest und im Erfolgsfall ist die Substitution *leer* (ϵ).

2.4.1 Algorithmus

Eine mögliche Implementierung der Unifikation [FH88] benutzt wechselseitig rekursive Hilfsfunktionen $\mathtt{d}$ und $\mathtt{D}$, die *nicht übereinstimmende* Teilbäume (*disagreement pair*) berechnet:

$$\text{FUN } \mathtt{d}\colon \mathtt{type} \times \mathtt{type} \to \mathtt{option}[\mathtt{type} \times \mathtt{type}]$$
$$\text{FUN } \mathtt{D}\colon \mathtt{seq}[\mathtt{type}] \times \mathtt{seq}[\mathtt{type}] \to \mathtt{option}[\mathtt{type} \times \mathtt{type}]$$

Die Funktion `d` liefert `nil`, falls die beiden Eingabeterme gleich sind. Ansonsten erfolgt eine – in beiden Termen simultane – Tiefensuche nach dem ersten Unterschied:

$$\begin{aligned}
\mathtt{d}(\mathtt{t}_1, \mathtt{t}_2) = \text{IF } & \mathtt{tvar}?(\mathtt{t}_1) \text{ THEN} \\
& \text{IF } \mathtt{tvar}?(\mathtt{t}_2) \text{ THEN} \\
& \quad \text{IF } \mathtt{var}(\mathtt{t}_1) = \mathtt{var}(\mathtt{t}_2) \text{ THEN } \mathtt{nil} \\
& \quad \text{ELSE } (\mathtt{t}_1, \mathtt{t}_2) \text{ FI} \\
& \text{ELSE } (\mathtt{t}_1, \mathtt{t}_2) \text{ FI} \\
& \text{ELSE IF } \mathtt{tvar}?(\mathtt{t}_2) \text{ THEN } (\mathtt{t}_1, \mathtt{t}_2) \\
& \quad \text{ELSE IF } \mathtt{id}(\mathtt{t}_1) = \mathtt{id}(\mathtt{t}_2) \wedge \#(\mathtt{args}(\mathtt{t}_1)) = \#(\mathtt{args}(\mathtt{t}_2)) \\
& \qquad \text{THEN } \mathtt{D}(\mathtt{args}(\mathtt{t}_1), \mathtt{args}(\mathtt{t}_2)) \\
& \qquad \text{ELSE } (\mathtt{t}_1, \mathtt{t}_2) \text{ FI FI FI}
\end{aligned}$$

Wenn die Toplevel-Konstruktoren (durch `id` selektiert) übereinstimmen, werden rekursiv die Argumente (durch `args` selektiert) solange paarweise verglichen, bis ein Unterschied entdeckt wird oder keiner vorliegt. Die Eingabe zur Funktion `D` sind immer zwei gleich lange Sequenzen, weil die Stelligkeit gleicher Konstruktoren gleich sein muss:

$$\begin{aligned}
& \mathtt{D}(\Diamond, \Diamond) = \mathtt{nil} \\
& \mathtt{D}(\mathtt{t}_1 :: \mathtt{r}_1, \mathtt{t}_2 :: \mathtt{r}_2) = \text{LET } \mathtt{p} = \mathtt{d}(\mathtt{t}_1, \mathtt{t}_2) \text{ IN} \\
& \qquad \text{IF } \mathtt{nil}?(\mathtt{p}) \text{ THEN } \mathtt{D}(\mathtt{r}_1, \mathtt{r}_2) \text{ ELSE } \mathtt{p} \text{ FI}
\end{aligned}$$

Die Unifikation sieht damit wie folgt aus:

$$\begin{aligned}
\mathtt{unify}(\mathtt{t}_1, \mathtt{t}_2) = \text{IF } & \mathtt{nil}?(\mathtt{d}(\mathtt{t}_1, \mathtt{t}_2)) \text{ THEN } \epsilon \quad \texttt{-} \textit{ erfolgreich fertig} \\
& \text{ELSE LET } (\mathtt{p}_1, \mathtt{p}_2) = \mathtt{d}(\mathtt{t}_1, \mathtt{t}_2) \text{ IN} \\
& \quad \mathtt{S} := \text{IF } \mathtt{tvar}?(\mathtt{p}_1) \wedge \mathtt{p}_1 \not\prec \mathtt{p}_2 \text{ THEN } [\mathtt{var}(\mathtt{p}_1) := \mathtt{p}_2] \\
& \qquad \text{IF } \mathtt{tvar}?(\mathtt{p}_2) \wedge \mathtt{p}_2 \not\prec \mathtt{p}_1 \text{ THEN } [\mathtt{var}(\mathtt{p}_2) := \mathtt{p}_1] \\
& \qquad \text{ELSE } \mathtt{nil} \text{ FI} ; \\
& \quad \mathtt{unify}(\mathtt{S}(\mathtt{t}_1), \mathtt{S}(\mathtt{t}_2)) \circ \mathtt{S} \\
& \text{FI}
\end{aligned}$$

Die Unifikation schlägt dann mit `nil` fehl, wenn keiner der beiden Teilterme $\mathtt{p}_1$ oder $\mathtt{p}_2$ eine Variable ist, also auf Grund ihrer Berechnung durch `d` die Toplevel-Konstruktoren verschieden sind. Liefert `d` eine Variable und einen Baum, dann muss im Prinzip die Variable durch diesen Baum ersetzt (substituiert) werden. Diese Ersetzung führt aber nur zum Ziel, wenn die Variable

im Baum selbst nicht vorkommt. Der *Occurs-Check* $\mathtt{var}(\mathtt{p}_1) \in \mathtt{tv}(\mathtt{p}_2)$ ist nichts anderes als der echte Teiltermtest $\mathtt{p}_1 \prec \mathtt{p}_2$; auf Grund der Monotonie von Substitutionen gilt dann $\mathtt{S}(\mathtt{p}_1) \prec \mathtt{S}(\mathtt{p}_2)$ und die Irreflexivität der echten Teiltermbeziehung verhindert eine Unifikation der Teilterme $(\mathtt{p}_1, \mathtt{p}_2)$ und damit der Gesamtterme $(\mathtt{t}_1, \mathtt{t}_2)$.

Es kann sein, dass sowohl $\mathtt{p}_1$ als auch $\mathtt{p}_2$ Variablen sind. In diesem Fall ist durch `d` gewährleistet, dass es sich um verschiedene Variablen handelt; ein *Occurs-Check* wäre dann nicht nötig und es ist egal, ob $\mathtt{p}_1$ zu $\mathtt{p}_2$ umbenannt wird oder umgekehrt.

Die Ein-Punkt-Substitution `S` wird durch den *Occurs-Check* auf jeden Fall idempotent und entfernt genau eine Variable aus den Termen $(\mathtt{S}(\mathtt{t}_1), \mathtt{S}(\mathtt{t}_2))$ für den deswegen terminierenden rekursiven Aufruf von `unify`. Das idempotente Gesamtergebnis ist im Erfolgsfall die sequenzielle Komposition von idempotenten Ein-Punkt-Substitutionen.

2.4.2 Komplexität

Der beschriebene Algorithmus ist eine Variante der Robinson-Unifikation [Rob71, CB83], die für einige Terme exponentiell aufwendig ist. Dazu betrachte man die folgenden beiden Terme mit je `n` Komponenten:

$$\mathtt{t}(\alpha_0, \ldots, \alpha_{\mathtt{n}-1})$$
$$\mathtt{t}(\mathtt{p}(\alpha_1, \alpha_1), \ldots, \mathtt{p}(\alpha_\mathtt{n}, \alpha_\mathtt{n}))$$

Als Ergebnis der Unifikation ergibt sich eine sequenzielle Komposition der Ein-Punkt-Substitutionen $[\alpha_{\mathtt{i}-1} := \mathtt{p}(\alpha_\mathtt{i}, \alpha_\mathtt{i})]$ für $(1 \leq \mathtt{i} \leq \mathtt{n})$, in Postfix-Notation:

$$[\alpha_0 := \mathtt{p}(\alpha_1, \alpha_1)] \ldots [\alpha_{\mathtt{n}-1} := \mathtt{p}(\alpha_\mathtt{n}, \alpha_\mathtt{n})]$$

Die Durchführung der Substitutionen bewirkt nun, dass α_0 für einen vollen binären Baum mit $2^\mathtt{n}$ Blättern $\alpha_\mathtt{n}$ steht und damit ein (finaler) Vergleich (im obigen Algorithmus durch `d`) entsprechend aufwendig ist. Ein solches Aufblähen der Terme kann nur vermieden werden, wenn die sequenziellen Substitutionen nie explizit appliziert werden. Für das obige Beispiel ist es möglich, nachdem durch `d` das erste Paar $(\alpha_0, \mathtt{p}(\alpha_1, \alpha_1))$ verarbeitet wurde, nur noch die restlichen Komponenten zu betrachten, in denen die Variable α_0 gar nicht mehr vorkommt. Allein diese Optimierung reicht nicht, um dieselben Terme mit Komponenten in umgekehrter (oder permutierter) Reihenfolge effizient zu unifizieren:

$\mathtt{t}(\alpha_{\mathtt{n}-1}, \ldots, \alpha_0)$
$\mathtt{t}(\mathtt{p}(\alpha_{\mathtt{n}}, \alpha_{\mathtt{n}}), \ldots, \mathtt{p}(\alpha_1, \alpha_1))$

Das erste Paar ist nun $(\alpha_{\mathtt{n}-1}, \mathtt{p}(\alpha_{\mathtt{n}}, \alpha_{\mathtt{n}}))$ und die entsprechende Substitution muss im Restterm, d.h. in der zweiten Komponente des zweiten Terms, berücksichtigt werden.

Die Idee der Martelli-Montanari-Unifikation [MM82] basiert nun darauf, zunächst alle nicht-übereinstimmenden Paare zu betrachten und dasjenige Paar zuerst zu behandeln, deren Variable – im Beispiel also α_0 – nicht im Rest vorkommt. Sollte es ein solches Paar nicht geben, dann liegt ein Zyklus vor. Besteht ein solcher Zyklus nur aus Variablen, dann sind diese Variablen alle gleich zu setzen; andernfalls signalisiert der Zyklus frühzeitig einen *Occurs*-Fehler, mit dem die Unifikation direkt abgebrochen werden kann.

Insgesamt ist die Unifikation als lineares Problem bekannt [PW78, dC86]. In der Praxis wird aber eigentlich immer die einfache Robinson-Unifikation verwendet, die für die meisten Fälle unschlagbar schnell ist. Unbrauchbar ist dieser Algorithmus nur für – absichtlich konstruierte – Extremfälle. In [ACF93] werden durchschnittliche Laufzeiten verglichen, bei denen *fast lineare* Implementierungen [MM82, RP89] gut mithalten. Das Haupthindernis für die Verwendung dieser Algorithmen sind aber die Konsequenzen für die Repräsentation der Terme. Statt (expandierte) Bäume müssen gerichtete Graphen verwaltet werden, die man sich wiederum als elementare Bäume zusammen mit sequenziellen Substitutionen für die gemeinsamen Teilbäume vorstellen kann. Der zusätzliche Verwaltungsaufwand betrifft also nicht nur die Unifikation, sondern auch alle anderen Komponenten, die Terme verarbeiten.

2.5 Algorithmus $\mathcal{M}$

Algorithmus $\mathcal{M}$ ist ein zu $\mathcal{W}$ alternativer Typinferenzalgorithmus, der praktische Vorteile hat. Während bei der Applikation $\mathtt{f}(\mathtt{e})$ der Algorithmus $\mathcal{W}$ den Typ des Arguments `e` unabhängig vom Typ der Funktion `f` berechnet und anschließend mit dem Argumenttyp von `f` unifiziert, fließt beim Algorithmus $\mathcal{M}$ der Argumenttyp von `f` direkt als Kontextwissen in die Typanalyse von `e` ein. Der *Folklore*-Algorithmus $\mathcal{M}$ [LY98] wird deswegen als *kontextsensitiv* und *top-down* bezeichnet:

$\mathtt{M}\colon \mathtt{env} \times \mathtt{expr} \times \mathtt{type} \times \mathtt{nat} \to \mathtt{option}[\mathtt{subst} \times \mathtt{nat}]$

Initial muss der Algorithmus $\mathcal{M}$ mit einer frischen Unbekannten als Eingabetyp aufgerufen werden. Der gesuchte prinzipale Typ ergibt sich dann aus dieser Unbekannten und der Ergebnissubstitution:

$\mathtt{M}(\mathtt{A},\mathtt{x},\mathtt{t},\mathtt{n}) =$
  IF $\mathtt{x} \in \mathtt{A}$
    THEN $\mathtt{S} := \mathtt{unify}(\mathtt{t}, \mathtt{inst}(\mathtt{n}, \mathtt{A}(\mathtt{x})));$
      $(\mathtt{S}, \mathtt{n} + \#(\mathtt{bv}(\mathtt{A}(\mathtt{x}))))$
    ELSE `nil` - *unbekannte Variable*
  FI

$\mathtt{M}(\mathtt{A},\mathtt{f}(\mathtt{e}),\mathtt{t},\mathtt{n}) =$
  $(\mathtt{S}_1, \mathtt{m}) := \mathtt{M}(\mathtt{A}, \mathtt{f}, \mathtt{tvar}(\mathtt{n}) \to \mathtt{t}, \mathtt{n}+1);$
  $(\mathtt{S}_2, \mathtt{l}) := \mathtt{M}(\mathtt{S}_1(\mathtt{A}), \mathtt{e}, \mathtt{S}_1(\mathtt{n}), \mathtt{m});$
  $(\mathtt{S}_2 \circ \mathtt{S}_1, \mathtt{l})$

$\mathtt{M}(\mathtt{A},\lambda\mathtt{x}.\mathtt{e},\mathtt{t},\mathtt{n}) =$
  $\mathtt{S}_1 := \mathtt{unify}(\mathtt{t}, \mathtt{tvar}(\mathtt{n}) \to \mathtt{tvar}(\mathtt{n}+1))$
  $(\mathtt{S}_2, \mathtt{m}) := \mathtt{M}(\mathtt{S}_1(\mathtt{A}) + \mathtt{x} : \mathtt{S}_1(\mathtt{n}), \mathtt{e}, \mathtt{S}_1(\mathtt{n}+1), \mathtt{n}+2)$
  $(\mathtt{S}_2 \circ \mathtt{S}_1, \mathtt{m})$

$\mathtt{M}(\mathtt{A},\text{LET } \mathtt{x}_1 = \mathtt{e}_1 \text{IN } \mathtt{e},\mathtt{t},\mathtt{n}) =$
  $(\mathtt{S}_1, \mathtt{m}) := \mathtt{M}(\mathtt{A}, \mathtt{e}_1, \mathtt{tvar}(\mathtt{n}), \mathtt{n}+1);$
  $(\mathtt{S}_2, \mathtt{l}) := \mathtt{M}(\mathtt{S}_1(\mathtt{A}) + \mathtt{x}_1 : \mathtt{gen}(\mathtt{S}_1(\mathtt{A}), \mathtt{S}_1(\mathtt{n})), \mathtt{e}, \mathtt{S}_1(\mathtt{t}), \mathtt{m});$
  $(\mathtt{S}_2 \circ \mathtt{S}_1, \mathtt{l})$

$\mathtt{M}(\mathtt{A}, \text{FIX } \mathtt{x}.\mathtt{e}, \mathtt{t}, \mathtt{n}) = \mathtt{M}(\mathtt{A} + \mathtt{x} : \mathtt{t}, \mathtt{e}, \mathtt{t}, \mathtt{n})$

Die Verwaltung des Zählers für die Generierung von frischen Unbekannten ist wieder offensichtlich. Bei Eingabe muss er größer als die Unbekannten in der Umgebung und im Eingabetyp sein, als Ausgabe ist er dann größer als alle (zusätzlichen) Unbekannten in der Ergebnissubstitution.

Algorithmus $\mathcal{M}$ benutzt Unifikation bei λ-Ausdrücken und Variablen, während Algorithmus $\mathcal{W}$ die Unifikation bei der Applikation aufrief.

Der praktische Vorteil des Algorithmus $\mathcal{M}$ ist, dass durch das top-down *ererbte* Kontextwissen Fehler früher aufdeckt werden können. Gemäß [LY98] terminiert der Algorithmus garantiert früher! Außerdem kann häufig die Fehlerursache genauer angegeben werden; noch besser könnten Fehler allerdings durch eine Kombination beider Algorithmen analysiert werden.

Die Konsistenz ähnelt nun der *Abgeschlossenheit* der Typableitungsrelation bezüglich Substitution:

$$\mathtt{M}(\mathtt{A}, \mathtt{e}, \mathtt{t}, \_) = (\mathtt{S}, \_) \;\Rightarrow\; \mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{S}(\mathtt{t})$$

Eine erfolgreiche Berechnung impliziert die Typableitbarkeit.

Die Vollständigkeit ist zusammen mit der Prinzipalität die Umkehrung der Konsistenz:

$$\mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{S}(\mathtt{t}) \;\Rightarrow\; \mathtt{M}(\mathtt{A}, \mathtt{e}, \mathtt{t}, \_) = (\mathtt{S}_0, \_) \;\wedge\; \exists \mathtt{R}.\ \mathtt{R}(\mathtt{S}_0(\mathtt{t})) = \mathtt{S}(\mathtt{t})$$

Wenn eine Typableitung möglich ist, dann ist die Berechnung mit $\mathcal{M}$ erfolgreich. Der Ergebnistyp $\mathtt{S}_0(\mathtt{t})$ ist allgemeiner als jeder ableitbare Typ $\mathtt{S}(\mathtt{t})$.

Für eine frische Unbekannte $\mathtt{n}$ und einen in $\mathtt{A}$ typisierbaren Ausdruck $\mathtt{e}$ berechnet

$$\mathtt{M}(\mathtt{A}, \mathtt{e}, \mathtt{tvar}(\mathtt{n}), \mathtt{n}+1) = (\mathtt{S}_0, \_)$$

den prinzipalen Typ $\mathtt{S}_0(\mathtt{n})$. Begründung: der allgemeinste Typ $\mathtt{t}_0$ ist definitionsgemäß allgemeiner als der berechnete Typ $\mathtt{S}_0(\mathtt{n})$, der wiederum auf Grund obiger Vollständigkeit allgemeiner als $\mathtt{S}(\mathtt{n})$ ist; wählt man nun für $\mathtt{S}$ die Substitution $[\mathtt{n} := \mathtt{t}_0]$, dann gilt $\mathtt{S}(\mathtt{n}) = \mathtt{t}_0$. Als Ungleichungskette ergibt sich:

$$\mathtt{t}_0 \preceq \mathtt{S}_0(\mathtt{n}) \preceq \mathtt{S}(\mathtt{n}) = \mathtt{t}_0$$

Die Typen $\mathtt{S}_0(\mathtt{n})$ und $\mathtt{t}_0$ sind also beide (gleich) prinzipal und können sich höchstens bezüglich der Benennung von Unbekannten unterscheiden.

Der Algorithmus $\mathcal{M}$ erlaubt das direkte *Testen* der Typableitung $\mathtt{A} \vdash \mathtt{e} : \mathtt{t}$, wenn also der Typ $\mathtt{t}$ *vorgegeben* ist. Die Ergebnissubstitution $\mathtt{S}_0$ kann dann höchstens Variablen umbenennen, da für $\mathtt{S}$ im obigen Vollständigkeitssatz die *leere Substitution* ϵ gewählt wird! Für explizite Typannotationen ist natürlich zu beachten, dass diese spezieller als *prinzipale* – also die ohne Typannotationen inferierten Typen – sein können. Dieser möglicherweise unbeabsichtigte Aspekt wird meistens ignoriert, könnte aber durchaus Teil einer ausführlicheren Diagnosemeldung sein.

2.6 Grenzen der polymorphen Typinferenz

Die polymorphe Typinferenz mit den Algorithmen $\mathcal{W}$ oder $\mathcal{M}$ ist geeignet für

- parametrische Polymorphie und
- Funktionen höherer Ordnung.

Die üblichen nicht-assoziativen Tupel, paralleles LET, wechselseitige Rekursion und Konstruktor-Muster (*Pattern*) sind Teil der Parsierung (*syntactic sugar*). *Überlagerung* wird nicht unterstützt!

Überlagerung würde nicht nur die Typinferenz im (VAR)-Fall in Frage stellen, sondern ebenso die syntaktischen Transformationen. Muster und Rekursionsstellen sind ohne Kontextwissen nicht mehr eindeutig erkennbar. Selbst α-Konversionen (und erst recht andere Reduktionen von λ-Ausdrücken) können nicht mehr rein syntaktisch durchgeführt werden, da der Bindungskontext nicht offensichtlich ist.

Eine konzeptionelle Beschränkung ist die *shallow*-Polymorphie; in der (ABS)-Regel wird deswegen der Typ der λ-Variablen nicht generalisiert. Eine Bindung von Typvariablen findet nur auf der äußersten Ebene der *Typschemata* statt, nicht innerhalb von *Typen*.

Im Vergleich dazu kann im mächtigeren Typsystem $\mathtt{F}_2$ von Girard und Reynolds [HM94] der Typ von λ-Variablen polymorph sein. Die Entscheidbarkeit der Typinferenz ist dafür allerdings nicht bekannt. Auch für FIX-Ausdrücke ist der Typ der rekursiven Funktion im Rumpf *monomorph*. Dadurch werden nicht-rekursive λ-Ausdrücke innerhalb von LETREC nicht immer optimal generalisiert (vgl. [Thi94] S. 285). Erlaubt man *polymorphe Rekursion* [Myc84] wie z.B. in HASKELL, dann wird die Typinferenz unentscheidbar [Hen91]. Polymorphe Rekursion oder polymorphe λ-Variablen sind allerdings dann kein Problem, wenn die polymorphen Typen explizit vorgegeben sind. Statt einer *Typinferenz* ist dann lediglich die *Typprüfung* von Applikationen mit bekannten polymorphen Funktionen nötig.

Die Motivation für polymorph rekursive Funktionen ergibt sich durch spezielle Datentypen; diese sind *nicht-regulär* [Hin99] aber in HASKELL und unorthogonalerweise auch in ML zulässig. Der folgende Datentyp `tree` repräsentiert einen *vollen binären Baum*[4] als Liste von Ebenen:

[4]Hinze [Hin99] gibt balancierte *Blätterbäume* an: `data Perfect a = a + Perfect(a × a)`

$$\begin{aligned}\text{TYPE } \texttt{tree}[\alpha] = \; & \texttt{empty} \\ & \texttt{cons}(\texttt{ft}: \alpha, \texttt{rt}: \texttt{tree}[\alpha \times \alpha])\end{aligned}$$

Jede Baumebene, die `rt`-Komponente vom Typ $\texttt{tree}[\alpha \times \alpha]$, besteht aus geschachtelten Paaren, um jeweils doppelt soviele Elemente (vom Typ α) aufnehmen zu können. Die Instanz vom leeren Listenende `empty` hängt also von der Baumhöhe ab; die Baumhöhe selbst müsste man analog zur Listenlänge definieren, allerdings *polymorph rekursiv*, was nur in HASKELL aber nicht in ML möglich ist.

Algebraisch – insbesondere in OPAL – ist obiger Datentyp `tree` unzulässig, da ein rekursiver und (mindestens) einstelliger Typkonstruktor (vgl. `seq` aus Abschnitt 1.1.1) gar nicht explizit appliziert wird und implizit immer die Applikation auf die formalen Parameter gemeint ist. Zusammenfassend lauten die Beschränkungen für die klassische Typinferenz wie folgt:

- shallow Polymorphie
- keine polymorphe Rekursion
- keine Überlagerung

Kapitel 3

Überlagerungsauflösung

In diesem Kapitel werden zwei altbekannte Algorithmen zur Überlagerungsauflösung vorgestellt, die sich bezüglich Effizienz und Verwaltungsaufwand unterscheiden. Der erste brute-force Algorithmus (Abschnitt 3.1) existiert in vier Varianten, die alle konzeptionell einfach aber im schlimmsten Fall exponentiell ineffizient sind. Die so genannte Cormack-Variante wird später für die Überlagerungsauflösung bei der Namensidentifikation benutzt. Der zweite und bekanntere Algorithmus (Abschnitt 3.2) verwaltet *Kandidaten* als Attribute an jedem Knoten und bleibt dadurch auch im schlimmsten Fall polynomial.

Die Notation ist wie zuvor „frei-funktional“, sie entspricht insofern nicht der typischen imperativen Darstellung dieser Algorithmen. Durch Überlagerungsauflösung wird üblicherweise kein zuvor unbekannter polymorpher Typ *inferiert*, sondern lediglich eine endliche Anzahl von vorgegebenen *monomorphen* Typen auf Konsistenz *geprüft*. Entsprechend unterscheiden sich die zugehörigen Sprachentwürfe. Im Hinblick auf die Verschmelzung mit der „uniformen Polymorphie“ (Kapitel 4) werden in Abschnitt 3.3 allein die „monomorphen Aspekte“ der Überlagerungsauflösung für λ- und monomorphe LET-Ausdrücke sowie Funktionen höherer Ordnung diskutiert.

Überlagerung bedeutet, dass ein und derselbe *Identifier* für verschiedene Operationen stehen kann. *Identifier* müssen hier von *Namen* unterschieden werden. Ein *Name* ist *eindeutig* und als Identifikationsnummer oder Zeiger auf eine konkrete Funktion vorstellbar. Ein *Identifier* dagegen kann mehrdeutig sein, d.h. verschiedene *Namen* und die damit eindeutig *benannten* Funktionen können mit *demselben Identifier bezeichnet* sein. Der Begriff *Namensüberlagerung* ist insofern verwirrend und wird hier vermieden; gemeint ist

Überlagerung der Funktions- oder Operationssymbole. Für *Überlagerungsauflösung* (*overload resolution*) ist außerdem der Begriff *Operator Identification* [PDM80] verbreitet.

Die Motivation für Überlagerung ist die Möglichkeit der *Wiederverwendung* aussagekräftiger Bezeichner. Umgekehrt erhöht Überlagerung die *Aussagekraft* von Bezeichnern, wenn diese für formal oder informal *ähnliche* Funktionen verwendet werden. Problematisch ist natürlich die ungeschickte oder (absichtlich) verwirrende Verwendung von Überlagerung; die üblichen Warnungen und Appelle an einen verantwortlichen Umgang mit *benutzerdefinierter* Überlagerung brauchen hier nicht wiederholt zu werden.

Überlagerung muss maschinell eindeutig auflösbar sein und ist nur sinnvoll, wenn diese Auflösung für den menschlichen Leser möglichst offensichtlich ist. Die wichtigste Information an einer Applikationsstelle ist dabei die Art und Anzahl der Argumente. Ansonsten kann noch der *umgebende Kontext* zur Identifizierung eines überlagerten Symbols beitragen.

Fast alle Programmiersprachen unterstützen Überlagerung, allerdings beschränkt auf die *eingebauten* arithmetischen Operationen für verschiedene Zahlenarten, z.B. `int`, `float` oder `double`. *Benutzerdefinierbare* Überlagerung ist z.B. in C++, JAVA, ADA und den algebraischen Sprachen zulässig.

Bottom-up Auflösbarkeit

Eine häufige (und durchaus vertretbare) Beschränkung der Überlagerung wird in [Wat90] *kontextunabhängig* genannt; bei ihr muss sich die Art oder Anzahl der Argumente von überlagerten Funktionen unterscheiden. Diese Beschränkung erlaubt eine einfache und effiziente Implementierung der Überlagerungsauflösung: *bottom-up* kann mit linearem Aufwand Überlagerung direkt aufgelöst und der Typ von Ausdrücken *synthetisiert* werden. Diese Beschränkung gilt z.B. für C++ und JAVA.

Ein Nachteil der *kontextunabhängigen* Überlagerung ist, dass insbesondere Konstanten (oder Variablen), also die *Blätter* der Ausdrücke, nicht überlagert werden können. Aus diesem Grund müssen in vielen Sprachen z.B. die Gleitkommakonstanten von ganzen Zahlen syntaktisch durch den Dezimalpunkt unterschieden werden.

Analog zur *kontextunabhängigen* (bottom-up) Überlagerungsauflösung könnte man Überlagerung auch auf reine top-down Auflösbarkeit beschränken. Die top-down Phase ist ausreichend, Überlagerungen ausgehend von einem

a-priori Typ direkt aufzulösen, wenn sich überlagerte Funktionen immer im Ergebnistyp oder der Stelligkeit unterscheiden. Eine derartige Beschränkung der Überlagerung ist für Programmiersprachen nicht verbreitet. Die überlagerten *homogenen* arithmetischen Operationen wären damit aber ebensogut aufzulösen.

Eine verbreitete Erweiterung der Überlagerungsauflösung ergibt sich durch implizite *Typkonversionen*; beispielsweise ist ein Ausdruck `1 + 2.0` nur deswegen typkorrekt, weil die ganze Zahl `1` implizit zur Gleitkommazahl `1.0` konvertiert wird. Diese Variation, die insbesondere Sprachen mit Subtypen oder objektorientierte Sprachen mit einer Klassenhierarchie (C++ oder JAVA) betrifft, wird hier nicht weiter untersucht. Bei einer *endlichen* Klassen- oder Subtyphierarchie kann Typkonversion als Überlagerungsauflösung betrachtet werden: für den obigen Ausdruck `1 + 2.0` könnte die Operation + auch inhomogen mit dem Typ $\texttt{int} \times \texttt{float} \rightarrow \texttt{float}$ überlagert sein.

3.1 Brute-Force Algorithmus

Die Überlagerung für *algebraische* Sprachen entspricht weitgehend der von ADA, bei der auch Funktionen überlagert werden dürfen, die sich nur im Resultatstyp unterscheiden. Die Implementierung der *kontextabhängigen* (oder kontextsensitiven) Überlagerungsauflösung für ADA wird in [MF91] als Herausforderung bezeichnet, die zu einer Vielzahl von Algorithmen führte, die sich hauptsächlich durch die Anzahl von abwechselnden top-down und bottom-up *Pässen* unterscheiden. Top-down wird ererbtes Kontextwissen benutzt, um Kandidaten auszuschließen, die keinen passenden Resultatstyp liefern und bottom-up werden Kandidaten ausgeschlossen, deren Argumenttypen nicht der synthetisierten Information entsprechen.

Der Ausgangspunkt sind Funktionen erster Ordnung mit null oder mehreren Argumenten und einem Ergebnis, also von der Form:

$$\texttt{f} : \texttt{t}_1 \times \ldots \times \texttt{t}_\texttt{k} \rightarrow \texttt{t} \qquad (\texttt{k} \geq 0)$$

Die (vereinfachte) Grammatik für Funktionsdeklarationen dazu sei:

$$\texttt{fct} ::== \texttt{ide} : \texttt{type}^* \rightarrow \texttt{type}$$

Die Typen sind einfache Bezeichner für Basistypen; mögliche Rekordtypen (oder Felder) muss man sich eindeutig *benannt* vorstellen. Selektoren, Kon-

struktoren und insbesondere Fallunterscheidungen sind auf jeden Fall *nichtgenerisch* und höchstens endlich oft überlagert.

Alle Ausdrücke sind bloße (verschachtelte) Applikationen von Funktionsbezeichnern mit folgender *Grammatik*:

`appl` ::== `ide`(`appl`, ..., `appl`)

Die Notation mit runden Klammern und die Trennung mehrerer Argumente durch Kommata dient lediglich der besseren Lesbarkeit. Eine ebensolche baumartige Klammerstruktur in Präfix-Notation entstünde durch die alternative Grammatik `appl` ::== (`ide` `appl`*). Die *Blätter* einer Applikation sind einfache Bezeichner `ide` *ohne Argumente*, d.h. Konstanten.

Ein ganzes Programm könnte eine Liste von Funktionsdeklarationen mit zugehörigen definierenden Ausdrücken `appl` sein, wobei die formalen Funktionsargumente $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k}$ für eine `k`-stellige Funktion besondere *lokale* Bezeichner `ide` sind, die nicht überlagert sein dürfen (siehe Abschnitt 3.3).

Der *Datentyp* zur Grammatik von `appl` (in OPAL-Notation) soll nun bezüglich der Bezeichner `ide` *parametrisiert* werden:

```
STRUCTURE Appl[ide]
  TYPE ide                                          - Typparameter
  TYPE appl == apply(id: ide, args: seq[appl])
```

Aufzulösende Ausdrücke können damit als `appl`[`ide`] und *vervollständigte Ausdrücke* durch `appl`[`fct`] modelliert werden. Alle potenziell überlagerten Bezeichner `ide` werden dabei durch die eindeutigen Funktionen `fct` ersetzt.

Als *Umgebung* betrachten wir eine endliche Liste von (globalen) Funktionen in beliebiger Reihenfolge. Da Bezeichner `ide` überlagert sein können, sind sie als Schlüssel für einen Tabellenzugriff auf den Typ *einer* Funktion ungeeignet[1]. Nur Bezeichner und Typ zusammen benennen eine Funktion eindeutig. Die Deklarationen überlagerter Funktionen sollten sich daher hinsichtlich ihrer Typisierung unterscheiden! Die Umgebung enthält dann keine Mehrfacheinträge für ein und dieselbe Funktion. Die Liste vom Typ `seq`[`fct`] repräsentiert insofern eine *Menge* von Funktionen.

Auch eine *Menge* von Auflösungsmöglichkeiten wird hier als Liste vom Typ `seq`[`appl`[`fct`]] verwaltet. Die Überlagerungsauflösung `R` (*resolve*) ist damit wie folgt typisiert:

[1] Als Tabellentyp für die Umgebung kommt höchstens `map`[`ide`, `seq`[`fct`]], nicht aber `map`[`ide`, `fct`] in Frage, wobei `fct` den Bezeichner `ide` redundant enthält.

FUN R: $\mathtt{seq}[\mathtt{fct}] \times \mathtt{appl}[\mathtt{ide}] \rightarrow \mathtt{seq}[\mathtt{appl}[\mathtt{fct}]]$

Aus einer globalen Umgebung B und einem k-stelligen Toplevel-*Symbol* c wird zunächst eine *Kandidatenmenge* C potenziell passender Toplevel-*Funktionen* extrahiert. Die Mengennotation dafür ist als simpler Filterausdruck über Listen zu lesen:

$$\mathtt{C} = \{\mathtt{f} \in \mathtt{B} \mid \mathtt{id}(\mathtt{f}) = \mathtt{c} \wedge \#(\mathtt{args}(\mathtt{f})) = \mathtt{k}\}$$

Für eine effiziente Berechnung der Kandidatenmenge C bietet sich eine Vorsortierung der Umgebung B bezüglich der Bezeichner an, etwa durch den Tabellentyp $\mathtt{map}[\mathtt{ide}, \mathtt{seq}[\mathtt{fct}]]$[2].

Für jedes der k Argumente $\mathtt{e_i}$ ($1 \leq \mathtt{i} \leq \mathtt{k}$) einer Applikation werden alle typkorrekten Vervollständigungen $\mathtt{E_i}$ *rekursiv* durch $\mathtt{R}(\mathtt{B}, \mathtt{e_i})$ berechnet.

Danach werden alle Kombinationen durchprobiert. Der Typ des i-ten Arguments einer Funktion $\mathtt{f} \in \mathtt{C}$ muss dabei identisch zum Resultatstyp der rekursiv berechneten Funktionsapplikation $\mathtt{f_i} \in \mathtt{E_i}$ an der i-ten Stelle sein, d.h. $\mathtt{arg_i}(\mathtt{f}) = \mathtt{res}(\mathtt{f_i})$:

$$\begin{array}{l}
\mathtt{R}(\mathtt{B},\mathtt{c}(\mathtt{e_1},\ldots,\mathtt{e_k})) = \\
\quad \text{LET } \mathtt{C} = \{\mathtt{f} \in \mathtt{B} \mid \mathtt{id}(\mathtt{f}) = \mathtt{c} \wedge \#(\mathtt{args}(\mathtt{f})) = \mathtt{k}\} \\
\qquad\quad \mathtt{E_1} = \mathtt{R}(\mathtt{B}, \mathtt{e_1}) \\
\qquad\quad \ldots \\
\qquad\quad \mathtt{E_k} = \mathtt{R}(\mathtt{B}, \mathtt{e_k}) \\
\quad \text{IN} \\
\quad \{\mathtt{f}(\mathtt{f_1},\ldots,\mathtt{f_k}) \mid \mathtt{f} \in \mathtt{C} \\
\quad \wedge\ \mathtt{f_1} \in \mathtt{E_1} \wedge \ldots \wedge \mathtt{f_k} \in \mathtt{E_k} \\
\quad \wedge\ \mathtt{arg_1}(\mathtt{f}) = \mathtt{res}(\mathtt{f_1}) \wedge \ldots \wedge \mathtt{arg_k}(\mathtt{f}) = \mathtt{res}(\mathtt{f_k})\}
\end{array}$$

Der Ausdruck der Form $\{\mathtt{e}(\mathtt{x})|\mathtt{c}(\mathtt{x})\}$ ist als *Listenkomprehension* [Thi94, PH+97] zu interpretieren. Ein Elementtest der Form $\mathtt{x} \in \mathtt{C}$ innerhalb der Konjunktion $\mathtt{c}(\mathtt{x})$ entspricht dabei einem *Generator* für die Werte zum Ausdruck $\mathtt{e}(\mathtt{x})$. Die Reihenfolge der Elemente in den Listen ist belanglos und wenn die initiale Umgebung B nur verschiedene Einträge enthält, dann sind auch die berechneten Vervollständigungen alle verschieden.

Offensichtlich ist nun, dass eine Applikation nur eindeutig typisierbar ist, wenn die berechnete Ergebnisliste einelementig ist. Eine leere Liste signalisiert einen Typfehler und eine mehrelementige Liste eine *Mehrdeutigkeit*.

[2]Mit einer primären Berücksichtigung der Stelligkeit könnte die Tabelle auch zweistufig sortiert vom Typ $\mathtt{map}[\mathtt{nat}, \mathtt{map}[\mathtt{ide}, \mathtt{seq}[\mathtt{fct}]]]$ vorliegen.

3.1.1 Beispiele

Der Ausdruck $\texttt{sin}(1 + 1)$ kann ohne Typkonversion und Dezimalpunkt eindeutig typisiert werden, wenn + nur homogen überlagert ist und `1` eine ganze oder eine Gleitkommazahl bezeichnet, aber `sin` eindeutig nur Gleitkommazahlen akzeptiert:

$$\begin{aligned}
&\texttt{1} : \{\texttt{int}, \texttt{float}\} \\
&+ : \{\texttt{int} \times \texttt{int} \to \texttt{int}, \texttt{float} \times \texttt{float} \to \texttt{float}\} \\
&\texttt{sin} : \texttt{float} \to \texttt{float}
\end{aligned}$$

$$\begin{aligned}
&\texttt{1} + \texttt{1} : \{\texttt{int}, \texttt{float}\} \\
&\texttt{sin}(\texttt{1} + \texttt{1}) : \texttt{float}
\end{aligned}$$

Die Überlagerung der Konstanten `1` ist also auflösbar und zwar selbst dann, wenn z.B. `sin` zusätzlich mit Typ $\texttt{double} \to \texttt{double}$ überlagert vorläge.

Bei Mehrdeutigkeiten kann offensichtlich die Auflösung eine exponentiell große Ausgabe erzeugen:

$$\begin{aligned}
&\texttt{c} : \{\texttt{t}_1, \texttt{t}_2\} \\
&\texttt{f} : \{\texttt{t}_1 \to \texttt{t}_1, \texttt{t}_1 \to \texttt{t}_2, \texttt{t}_2 \to \texttt{t}_1, \texttt{t}_2 \to \texttt{t}_2\}
\end{aligned}$$

Für den Ausdruck `c` gibt es 2 Lösungen, 4 für $\texttt{f}(\texttt{c})$ und 8 für f(f(c)). Für den Ausdruck $\texttt{f}(\ldots(\texttt{f}(\texttt{c}))\ldots)$ mit `n` Vorkommen von `f` ergeben sich $2^{\texttt{n}+2}$ Lösungen. Dieser Ausdruck wird später als $\texttt{f}^{\texttt{n}}(\texttt{c})$ abgekürzt, obwohl diese *Metanotation* eher die `n`-malige Applikation *derselben* Funktion suggeriert als `n` aufeinanderfolgende Applikationen von *überlagerten* Funktionen `f`.

Explodierende Zwischenergebnisse können leider auch entstehen, wenn ein Ausdruck insgesamt eindeutig typisierbar ist. Dazu betrachte man die zusätzlichen Deklarationen $\texttt{c} : \texttt{t}_3, \texttt{f} : \texttt{t}_3 \to \texttt{t}_3$ und $\texttt{g} : \texttt{t}_3 \to \texttt{t}_3$ für den Ausdruck $\texttt{g}(\texttt{f}^{\texttt{n}}(\texttt{c}))$:

$$\begin{aligned}
&\texttt{c} : \{\texttt{t}_1, \texttt{t}_2, \texttt{t}_3\} \\
&\texttt{f} : \{\texttt{t}_1 \to \texttt{t}_1, \texttt{t}_1 \to \texttt{t}_2, \texttt{t}_2 \to \texttt{t}_1, \texttt{t}_2 \to \texttt{t}_2, \texttt{t}_3 \to \texttt{t}_3\} \\
&\texttt{g} : \texttt{t}_3 \to \texttt{t}_3
\end{aligned}$$

Die weiteren Deklarationen vergrößern höchstens die Ergebnismenge für den Teilausdruck $\texttt{f}^{\texttt{n}}(\texttt{c})$. Im Kontext von `g` sind allerdings die vielen Lösungen für $\texttt{f}^{\texttt{n}}(\texttt{c})$ mit Ergebnistyp $\texttt{t}_1$ oder $\texttt{t}_2$ völlig irrelevant und eine naheliegende Optimierung ist es, den von `g` *erwarteten* Typ $\texttt{t}_3$ bei der Berechnung von $\texttt{f}^{\texttt{n}}(\texttt{c})$ zu berücksichtigen. Das Beispiel wird dadurch sofort linear auflösbar, da immer genau eine Funktion `f` zum Typ $\texttt{t}_3$ passt.

3.1.2 Varianten

Statt rekursiv alle Lösungen $\mathtt{E_i}$ für die Teilausdrücke $\mathtt{e_i}$ durch $\mathtt{R(B, e_i)}$ *unabhängig* zu berechnen, kann man auch für jeden Toplevel-Kandidaten $\mathtt{f} \in \mathtt{C}$ die Mengen $\mathtt{E_i}$ *abhängig* vom Typ $\mathtt{arg_i(f)}$ berechnen. Der Algorithmus erhält dafür ein zusätzliches Argument vom Typ `type`. Für den ersten Aufruf übergibt man dabei den so genannten *a-priori* Typ, der für einen definierenden Ausdruck der Ergebnistyp aus der zugehörigen Funktionsdeklaration ist:

$$\begin{aligned}\mathtt{R(B,c(e_1,\ldots,e_k),t)} = \{\mathtt{f(f_1,\ldots,f_k)} \mid\ & \mathtt{f} \in \mathtt{B} \\ & \wedge\ \mathtt{id(f)} = \mathtt{c} \wedge \mathtt{\#(args(f))} = \mathtt{k} \wedge \mathtt{res(f)} = \mathtt{t} \\ & \wedge\ \mathtt{f_1} \in \mathtt{R(B, e_1, arg_1(f))} \wedge \ldots \wedge \mathtt{f_k} \in \mathtt{R(B, e_k, arg_k(f))}\end{aligned}$$

Auf diese Weise werden alle legalen Möglichkeiten *top-down* durchprobiert, wobei für den Ausdruck $\mathtt{g(f^n(c))}$ die falschen Kandidaten für `f` direkt ausgeschlossen werden. Diese Variante stammt ursprünglich (imperativ notiert in [MF91]) von G.V. Cormack 1981.

Im Vergleich zum vorherigen bottom-up Algorithmus ist die Cormack-Variante nicht immer effizienter. Selbst wenn eine getrennte Berechnung für *jeden* Typ ebenso aufwendig ist wie eine einzige Berechnung für *alle* Typen, dann erfolgt z.B. für den bloßen Ausdruck $\mathtt{f^n(c)}$ in der Cormack-Variante für *jeden* (der vier!) Toplevel-Kandidaten $\mathtt{f} \in \mathtt{C}$ (und nicht nur für jeden der beiden Typen!) ein rekursiver Aufruf, wobei jeweils für die Typen $\mathtt{t_1}$ und $\mathtt{t_2}$ zweimal dasselbe berechnet wird.

Für einen effizienten Zugriff bei der typabhängigen Berechnung von Kandidaten bietet sich sogar eine dreistufige Vorsortierung bezüglich Stelligkeit, Bezeichner und Ergebnistyp an: `map`[`nat`, `map`[`ide`, `map`[`type`, `seq`[`fct`]]]]

Als weitere Variante wird in [MF91] vorgeschlagen, eine typabhängige Berechnung nur für den Spezialfall durchzuführen, wenn für alle Toplevel-Kandidaten $\mathtt{f} \in \mathtt{C}$ der Typ an der `i`-ten Argumentposition *eindeutig* ist. Dieses ist insbesondere dann der durchaus häufige Fall, wenn die Kandidatenmenge `C` einelementig ist oder (unabhängig von der Stelligkeit) das Toplevel-Symbol gar nicht überlagert ist. Bei verschiedenen Typen für $\mathtt{arg_i(f)}$ werden wie zuvor die Mengen $\mathtt{E_i}$ *typunabhängig* berechnet und Doppelberechnungen vermieden; der Ausdruck $\mathtt{g(f^n(c))}$ bleibt effizient auflösbar.

Der ursprüngliche bottom-up Algorithmus `R` kann leicht mit einem zusätzlichen Eingabeparameter vom Typ `option`[`type`] modifiziert werden. Falls der *optionale* Typ `t` vorliegt, werden nur Kandidaten $\mathtt{f} \in \mathtt{C}$ mit passendem Ergebnistyp $\mathtt{t} = \mathtt{res(f)}$ berücksichtigt:

$$
\begin{array}{l}
\mathtt{R(B,c(e_1,\ldots,e_k),t)} = \\
\quad \text{LET } \mathtt{C} = \{\mathtt{f} \in \mathtt{B} \mid \mathtt{id(f)} = \mathtt{c} \wedge \mathtt{\#(args(f))} = \mathtt{k} \wedge \\
\qquad\qquad\qquad (\mathtt{t} = \mathtt{nil} \vee \mathtt{t} = \mathtt{res(f)})\} \\
\qquad \ldots
\end{array}
$$

Für die rekursiven Aufrufe an den $\mathtt{k}$ Argumenten $\mathtt{e_i}$ müssen folgende Typmengen $\mathtt{T_i}$ für $\mathtt{1} \leq \mathtt{i} \leq \mathtt{k}$ untersucht werden:

$$\mathtt{T_i} = \{\mathtt{arg_i(f)} \mid \mathtt{f} \in \mathtt{C}\}$$

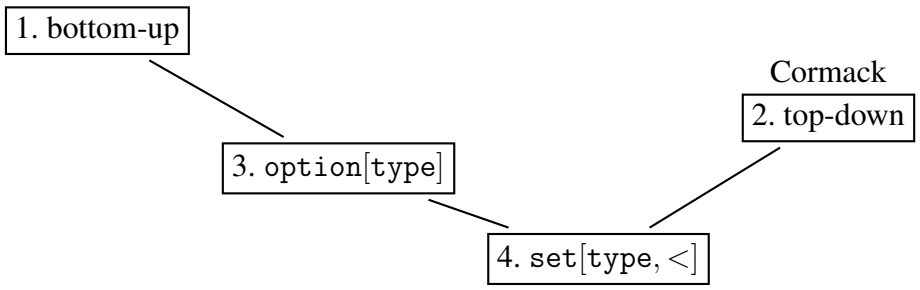
Die Typmengen $\mathtt{T_i}$ sind nun *echte Mengen* vom Typ $\mathtt{set[type}, <]$, die mit Hilfe einer $<$-Relation vom Typ $\mathtt{type} \times \mathtt{type} \rightarrow \mathtt{bool}$ keine Elemente doppelt enthalten. Falls eine Typmenge $\mathtt{T_i}$ einelementig ist, dann kann der eindeutige Typ $\mathtt{t_i} \in \mathtt{T_i}$ bei der Rekursion am $\mathtt{i}$-ten Argument berücksichtigt werden, ansonsten wird $\mathtt{nil}$ übergeben:

$$
\begin{array}{l}
\text{LET } \mathtt{t_i} = \text{IF } \mathtt{\#(T_i)} = \mathtt{1} \text{ THEN } \mathtt{min(T_i)} \text{ ELSE } \mathtt{nil} \text{ FI} \\
\text{IN } \mathtt{E_i} = \mathtt{R(B, e_i, t_i)}
\end{array}
$$

Die Rekursionsstrategie entspricht der ursprünglichen bottom-up Variante. Nimmt man aber z.B. die irrelevante Überlagerung $\mathtt{g} : \mathtt{t_4} \rightarrow \mathtt{t_4}$ zur Umgebung hinzu, dann wird der obige Ausdruck $\mathtt{g(f^n(c))}$ wieder wie in der bottom-up Variante ineffizient aufgelöst, während die Cormack-Variante dafür effizient bleibt! Eine andere offensichtliche Variante ist also, direkt die gesamte Typmenge $\mathtt{T_i}$ (vom Typ $\mathtt{set[type}, <]$) als zusätzliches Argument zu übergeben:

$$
\begin{array}{l}
\mathtt{R(B,c(e_1,\ldots,e_k),T)} = \\
\quad \text{LET } \mathtt{C} = \{\mathtt{f} \in \mathtt{B} \mid \mathtt{id(f)} = \mathtt{c} \wedge \mathtt{\#(args(f))} = \mathtt{k} \wedge \mathtt{res(f)} \in \mathtt{T}\} \\
\qquad \mathtt{T_1} = \{\mathtt{arg_1(f)} \mid \mathtt{f} \in \mathtt{C}\} \\
\qquad \ldots \\
\qquad \mathtt{T_k} = \{\mathtt{arg_k(f)} \mid \mathtt{f} \in \mathtt{C}\} \\
\qquad \mathtt{E_1} = \mathtt{R(B, e_1, T_1)} \\
\qquad \ldots \\
\qquad \mathtt{E_k} = \mathtt{R(B, e_k, T_k)} \\
\quad \text{IN } \ldots
\end{array}
$$

Der Aufwand für die Verwaltung von Typmengen als Eingabeargument wird in [MF91] für die Praxis als zu hoch in Relation zum Nutzen bewertet und das *optionale* Typargument bevorzugt. Im Vergleich zur Cormack-Variante vermeidet die Typmengenbildung aber genau unnötige Doppelberechnungen. Insgesamt ist approximativ (ohne Verwaltungsaufwand) die vierte und letzte Variante für alle Beispiele am besten. Die top-down Cormack-Variante dagegen kann besser oder schlechter als die anderen Varianten sein:



Für alle vier brute-force Varianten kann das Laufzeitverhalten exponentiell sein! Als letztes Beispiel dazu betrachte man den Ausdruck $\mathtt{h}(\mathtt{f}^{\mathtt{n}}(\mathtt{c}), \mathtt{b})$ mit folgender Umgebung:

$\mathtt{b} : \mathtt{t}_2$
$\mathtt{c} : \{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3\}$
$\mathtt{f} : \{\mathtt{t}_1 \to \mathtt{t}_1, \mathtt{t}_1 \to \mathtt{t}_2, \mathtt{t}_2 \to \mathtt{t}_1, \mathtt{t}_2 \to \mathtt{t}_2, \mathtt{t}_3 \to \mathtt{t}_3\}$
$\mathtt{h} : \{\mathtt{t}_3 \times \mathtt{t}_2 \to \mathtt{t}_3, \mathtt{t}_1 \times \mathtt{t}_1 \to \mathtt{t}_3\}$

Für die überlagerten Funktionen $\mathtt{h}$ mit den Typen $\mathtt{t}_3 \times \mathtt{t}_2 \to \mathtt{t}_3$ und $\mathtt{t}_1 \times \mathtt{t}_1 \to \mathtt{t}_3$ und der Konstanten $\mathtt{b} : \mathtt{t}_2$ ist $\mathtt{h}(\mathtt{f}^{\mathtt{n}}(\mathtt{c}), \mathtt{b})$ eindeutig, da das zweite Argument $\mathtt{b}$ garantiert, dass für das erste Argument $\mathtt{f}^{\mathtt{n}}(\mathtt{c})$ nur der eindeutige Kandidat vom Typ $\mathtt{t}_3$ in Frage kommt. Allein die Typmenge $\mathtt{T}_1 = \{\mathtt{arg}_1(\mathtt{h})\} = \{\mathtt{t}_1, \mathtt{t}_3\}$ bildet aber keine ausreichende Beschränkung für eine effiziente Berechnung von $\mathtt{f}^{\mathtt{n}}(\mathtt{c})$.

3.2 Attributierungsalgorithmus

Pathologische Mehrdeutigkeiten lassen sich besser in den Griff bekommen, wenn Kandidaten nicht direkt zu Teilausdrücken kombiniert werden, sondern getrennt für jeden Knoten berechnet werden. Bei dieser verbreiteten Überlagerungsauflösung [WM92] werden bottom-up Kandidaten mit passenden Argumenttypen ausgewählt und deren Ergebnistypen nach oben *synthetisiert* und top-down werden die Argumenttypen von Kandidaten mit passenden Ergebnistypen nach unten *vererbt*. Der *attributierte* Syntaxbaum wird hier funktional als `appl[seq[fct]]` verwaltet. Initial werden an jedem Knoten alle Kandidaten berücksichtigt, die rein syntaktisch nur bezüglich der Bezeichnung und Stelligkeit passen:

FUN $\texttt{init}: \texttt{seq}[\texttt{fct}] \times \texttt{appl}[\texttt{ide}] \to \texttt{appl}[\texttt{seq}[\texttt{fct}]]$

$\texttt{init}(\texttt{B}, \texttt{c}(\texttt{e}_1, \ldots, \texttt{e}_k)) =$
LET $\texttt{C} = \{\texttt{f} \in \texttt{B} \mid \texttt{id}(\texttt{f}) = \texttt{c} \wedge \#(\texttt{args}(\texttt{f})) = \texttt{k}\}$
$\quad \texttt{E}_1 = \texttt{init}(\texttt{B}, \texttt{e}_1)$
$\quad \ldots$
$\quad \texttt{E}_k = \texttt{init}(\texttt{B}, \texttt{e}_k)$
IN $\texttt{C}(\texttt{E}_1, \ldots \texttt{E}_k)$ - *appl-Konstruktion*

Bottom-up werden solche Kandidaten f ausgeschlossen, deren i-ter Argumenttyp $\texttt{arg}_\texttt{i}(\texttt{f})$ nicht durch (untere) Kandidaten aus $\texttt{A}_\texttt{i}$ erreichbar ist, d.h. für die $\texttt{arg}_\texttt{i}(\texttt{f}) \neq \texttt{res}(\texttt{f}_\texttt{i})$ für alle $\texttt{f}_\texttt{i} \in \texttt{A}_\texttt{i}$ ist:

FUN $\texttt{bottomUp}: \texttt{appl}[\texttt{seq}[\texttt{fct}]] \to \texttt{appl}[\texttt{seq}[\texttt{fct}]]$

$\texttt{bottomUp}(\texttt{C}(\texttt{E}_1, \ldots, \texttt{E}_k)) =$
LET $\texttt{A}_1 = \texttt{bottomUp}(\texttt{E}_1)$
$\quad \ldots$
$\quad \texttt{A}_k = \texttt{bottomUp}(\texttt{E}_k)$
$\quad \texttt{T}_1 = \{\texttt{res}(\texttt{f}_1) \mid \texttt{f}_1 \in \texttt{A}_1\}$
$\quad \ldots$
$\quad \texttt{T}_k = \{\texttt{res}(\texttt{f}_k) \mid \texttt{f}_k \in \texttt{A}_k\}$
$\quad \texttt{R} = \{\texttt{f} \in \texttt{C} \mid \texttt{arg}_1(\texttt{f}) \in \texttt{T}_1 \wedge \ldots \wedge \texttt{arg}_k(\texttt{f}) \in \texttt{T}_k\}$
IN $\texttt{R}(\texttt{A}_1, \ldots \texttt{A}_k)$

Top-down werden solche Kandidaten $\texttt{f}_\texttt{i}$ an den Argumentknoten ausgeschlossen, für die es *kein* (oberes) f gibt mit $\texttt{arg}_\texttt{i}(\texttt{f}) = \texttt{res}(\texttt{f}_\texttt{i})$. Die möglichen Argumenttypen werden als zusätzlicher Parameter vom Typ $\texttt{set}[\texttt{type}, <]$ übergeben:

FUN $\texttt{topDown}: \texttt{set}[\texttt{type}, <] \times \texttt{appl}[\texttt{seq}[\texttt{fct}]] \to \texttt{appl}[\texttt{seq}[\texttt{fct}]]$

$\texttt{topDown}(\texttt{T}, \texttt{C}(\texttt{E}_1), \ldots, \texttt{E}_k)) =$
LET $\texttt{R} = \{\texttt{f} \in \texttt{C} \mid \texttt{res}(\texttt{f}) \in \texttt{T}\}$
$\quad \texttt{T}_1 = \{\texttt{arg}_1(\texttt{f}) \mid \texttt{f} \in \texttt{R}\}$
$\quad \ldots$
$\quad \texttt{T}_k = \{\texttt{arg}_k(\texttt{f}) \mid \texttt{f} \in \texttt{R}\}$
$\quad \texttt{A}_1 = \texttt{topDown}(\texttt{T}_1, \texttt{E}_1)$
$\quad \ldots$
$\quad \texttt{A}_k = \texttt{topDown}(\texttt{T}_k, \texttt{E}_k)$
IN $\texttt{R}(\texttt{A}_1, \ldots \texttt{A}_k)$

Der Ausdruck

$$\texttt{topDown}(\{\texttt{t}\}, \texttt{bottomUp}(\texttt{init}(\texttt{B}, \texttt{e})))$$

mit dem a-priori Typ t und der Umgebung B liefert für die Applikation e eine Kandidatenbeschreibung, die eindeutig nur dann ist, wenn sämtliche Mengen an den Knoten einelementig sind. Leere Kandidatenlisten signalisieren Typfehler und mehrelementige Mehrdeutigkeiten. Dieser Auflösungsalgorithmus in [WM92] stammt ursprünglich aus [PDM80].

Diagnosemeldungen im Fehlerfall sowie Korrektheitsfragen sollen hier nicht weiter erläutert werden. Im Beispiel $\texttt{h}(\texttt{f}^{\texttt{n}}(\texttt{c}), \texttt{b})$ sorgt die bottom-up Phase dafür, dass h durch b eindeutig wird; die anschließende top-down Phase propagiert den eindeutigen Typ $\texttt{t}_3$ dann durch den Teilausdruck $\texttt{f}^{\texttt{n}}(\texttt{c})$, der damit ebenfalls eindeutig wird.

Das (naheliegende) Vertauschen der Durchgänge ist nicht ausreichend, da erst die bottom-up Phase den Kandidaten h an der Wurzel eindeutig macht und ohne eine nachfolgende top-down Phase diese Information im Teilausdruck $\texttt{f}^{\texttt{n}}(\texttt{c})$ fehlt. Beginnt man also mit einer top-down Phase sind drei Durchgänge (top-down, bottom-up, top-down) erforderlich, wobei die ersten beiden Phasen (und die Initialisierung) leicht zu *einer* rekursiv absteigenden Funktion zusammengefasst werden können. Dadurch entsteht eine komplexere bottom-up und Initialisierungsphase mit i.d.R. kleineren Kandidatenmengen.

Aus dem Ergebnis appl[seq[fct]] am Ende ergibt sich (bei Eindeutigkeit) natürlich leicht der aufgelöste Ausdruck appl[fct] (oder eine Fehlermeldung).

Die Überlagerungsauflösung mit Attributen ist polynomial. Alle Mengen sind nie größer als die Maximalzahl o für eine Überlagerung. An jedem der n Knoten müssen diese Mengen für k Argumente untersucht werden, d.h. der Aufwand ist von der Ordnung $\texttt{O}(\texttt{okn})$. Der zusätzliche Verwaltungsaufwand bewirkt im Vergleich zur direkten Auflösung, dass auch pathologische Fälle polynomial analysiert werden können, der Aufwand für nicht-pathologische Fälle ist dafür im Durchschnitt größer.

3.3 Erweiterungen

In der obigen Überlagerungsauflösung wurden nur Applikationen und Funktionen erster Ordnung betrachtet. Alle Funktionen wurden durch die Angabe einer Typsignatur $\texttt{f} : \texttt{t}_1 \times \ldots \times \texttt{t}_\texttt{k} \rightarrow \texttt{t}$ *deklariert*. Für eine *Funktionsdefinition*

$\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_k) = \mathtt{e}$ sind dann die Typen der Argumente und der Ergebnistyp bekannt. Die formalen Argumente $\mathtt{x}_i : \mathtt{t}_i$ werden wie Konstanten ein Teil der *globalen* Umgebung zur Auflösung des definierenden Ausdrucks `e`; der Ergebnistyp `t` von `f` ist der a-priori Typ von `e`. Die Analyse (wechselseitig) rekursiver Funktionen ist völlig unproblematisch, da die Typsignaturen bekannt sind. Komplikationen entstehen nur, wenn Teile der Typinformation fehlen.

3.3.1 Inferenz von fehlender Typinformation

Eine alternative Form für *nicht-rekursive* Funktionsdefinitionen wäre z.B. die Notation $\mathtt{f}(\mathtt{x}_1 : \mathtt{t}_1, \ldots, \mathtt{x}_k : \mathtt{t}_k) = \mathtt{e}$, bei der der Ergebnistyp nicht gegeben ist, sondern dieser sich allein aus der bottom-up Analyse des definierenden Ausdrucks `e` ergeben muss. Erst für *nachfolgende* Applikationen ist dann der Typ von `f` bekannt. Unüblich ist es, *nachfolgende Applikationen* von `f` als Beschränkungen für die Typanalyse von `e` zu berücksichtigen.

Es ist ebenfalls nicht leicht möglich, auf die Typinformation $\mathtt{t}_i$ für ein Argument $\mathtt{x}_i$ zu verzichten. Ein Typ $\mathtt{t}_i$ könnte sich aus den *aktuellen* `i`-ten Argumenten der *nachfolgenden Applikationen* von `f` ergeben *oder aus der Benutzung von* $\mathtt{x}_i$ *im definierenden Ausdruck* `e`. Die bei der top-down Analyse erzeugte Typmenge für den Teilausdruck $\mathtt{x}_i$ könnte z.B. einelementig sein. Kommt $\mathtt{x}_i$ in `e` *mehrfach* vor, dann könnte die Schnittmenge der top-down berechneten Typmengen für diese Vorkommen untersucht werden. Im schlimmsten Fall können die möglichen Typen für die Argumente $\mathtt{x}_i$ einfach (bottom-up) durchprobiert werden. (Der Aufwand bliebe polynomial.)

Eine Inferenz von Typen *rekursiver* Funktionen (der Form FIX `f.e`) findet i.d.R. nicht statt; Funktionstypen müssen durch FIX $\mathtt{f} : \mathtt{t}_1 \times \ldots \times \mathtt{t}_k \to \mathtt{t}$.`e` *global* vorgegeben werden.

3.3.2 Lambda-Ausdrücke

Für einen λ-Ausdruck $\lambda\mathtt{x}.\mathtt{e}$ ist weder der Typ des Arguments `x` noch der Ergebnistyp bekannt, d.h. diese Typen müssen sich aus der Analyse des Ausdrucks `e` *oder aus der Applikationsstelle des λ-Ausdrucks* ergeben.

Zum Beispiel könnte sich der Argumenttyp für `x` in der Applikation $(\lambda\mathtt{x}.\mathtt{e})(\mathtt{a})$ durch den Typ von `a` ergeben; bei einer Überlagerung von `a` würde die Menge der Typen für `x` zumindest beschränkt. Der Ergebnistyp des λ-Ausdrucks

wird z.B. durch eine umfassende Applikation $\mathtt{f}((\lambda \mathtt{x.e})(\mathtt{a}))$ beschränkt und diese Typinformation kann als a-priori Typ (oder Typmenge) in die Typanalyse von e einfließen.

Bei einer Gleichbehandlung von Ausdrücken der Form $\lambda \mathtt{x_1} : \mathtt{t_1}, \ldots, \mathtt{x_k} : \mathtt{t_k}.\mathtt{e}$ und den benannten Funktionen würde Information aus umfassenden Applikationen genau nicht in die Überlagerungsauflösung einfließen. Allerdings unterscheiden sich die anonymen λ-Ausdrücke von benannten Funktionen auch dadurch, dass sie (per Notation) genau *einmal* appliziert werden.

In PVS wird die *eindeutige* Typisierung von λ-Variablen verlangt. Die λ-Ausdrücke müssen also von der Form $\lambda \mathtt{x_1} : \mathtt{t_1}, \ldots, \mathtt{x_k} : \mathtt{t_k}.\mathtt{e}$ mit unterschiedlichen $\mathtt{x_i}$ sein und die Information $\mathtt{x_i} : \mathtt{t_i}$ wird (überschreibend) zur Umgebung für die Typisierung von e hinzugefügt. (Aus dem umfassenden Kontext berechnet PVS top-down eine beschränkende a-priori Typmenge für den Ergebnistyp von e.)

Die Problematik der Überlagerungsauflösung eines *ungetypten* λ-Ausdrucks $\lambda \mathtt{x.e}$ wird im Zusammenhang mit der algebraischen Typanalyse im Folgekapitel 4 behandelt.

Einen Sonderfall bilden die *monomorphen* LET-Ausdrücke $\mathtt{LET}\ \mathtt{x_1} = \mathtt{e_1}\ \mathtt{IN}\ \mathtt{e}$. Der Typ für $\mathtt{x_1}$ braucht (in PVS) nicht mehr angegeben werden, sondern muss sich aus der Typberechnung für $\mathtt{e_1}$ ergeben. Die Transformation des LET-Ausdrucks in einen äquivalenten λ-Ausdruck $(\lambda \mathtt{x_1} : \mathtt{typeOf}(\mathtt{e_1}).\mathtt{e})(\mathtt{e_1})$ ist also nicht mehr rein syntaktisch möglich sondern erfordert Typanalyse. Die Typberechnung $\mathtt{typeOf}(\mathtt{e_1})$ ist dabei *unabhängig* von den Vorkommen der Variablen $\mathtt{x_1}$ im Ausdruck e, die den Typ von $\mathtt{e_1}$ *zusätzlich* beschränken könnten. Die Analyse eines ungetypten λ-Ausdrucks wird so (in PVS) vermieden.

3.3.3 Funktionen höherer Ordnung

Welche Zusatzkomplikationen sind bei Überlagerungsauflösung für Funktionen höherer Ordnung zu beachten? Die bisherige Überlagerungsauflösung ist unabhängig von der Menge der Basistypen. So konnten Rekordtypen oder Felder wie Basistypen behandelt werden; das ist ebenso für Funktionstypen möglich! Insbesondere brauchen die Typen an *Argumentpositionen* nicht tiefer analysiert, sondern nur verglichen werden. Hinzu kommt, dass sich die *Ordnung* eines Funktionstyps aus der maximalen Schachtelungstiefe des Funktionstypkonstruktors $\rightarrow$ in *Argumentpositionen* ergibt [Thi94] (S. 61). Ein

Funktionstyp im Ergebnis ist lediglich die *curried* Schreibweise, die *unterversorgte* Applikationen ermöglicht.

Für die Überlagerungsauflösung bedeutet das, dass die *Stelligkeit* einer Funktion `f` vom Typ $\mathtt{t_1} \to \ldots \mathtt{t_k} \to \mathtt{t}$ in einer Applikation variieren kann. Die Funktion `f` ist aber nur von höherer Ordnung, wenn mindestens ein Argumenttyp $\mathtt{t_i}$ ein Funktionstyp ist. Wenn `t` kein Funktionstyp ist, ist `f` maximal `k`-stellig! Man kann sich dann `f` auf $\mathtt{k}+1$ Weisen überlagert vorstellen und zwar null- bis `k`-stellig. Eine unterversorgte `i`-stellige ($\mathtt{i} < \mathtt{k}$) Applikation $\mathtt{f}(\mathtt{e_1}) \ldots (\mathtt{e_i})$ liefert einen Funktionstyp mit $\mathtt{k} - \mathtt{i}$ restlichen Argumenten.

Die bei Funktionen erster Ordnung zur Konstruktion von komplexeren Datenstrukturen unerlässlichen *Tupeltypen* können wie in ML durch Funktionstypen simuliert werden und bedürfen keiner Extrabehandlung.

Die *Blätter* der *linksassoziativ* geklammerten Applikationen von Core-ML sind einfache Funktionssymbole, λ- und Fixpunktausdrücke, die bei der Überlagerungsauflösung die Rollen der *Toplevel-Kandidaten* übernehmen. Monomorphe LET-Ausdrücke werden in ihre äquivalenten, explizit getypten λ-Ausdrücke transformiert.

Zum Beispiel ist der geschachtelte λ-Ausdruck $\lambda\mathtt{x_1} : \mathtt{t_1}.\lambda\mathtt{x_2} : \mathtt{t_2}.\mathtt{e}$ eine anonyme Funktionsdefinition vom Typ $\mathtt{t_1} \to \mathtt{t_2} \to \mathtt{t}$, wobei sich der Typ `t` aus dem Kontext und der Typanalyse von `e` mit der um $\mathtt{x_1} : \mathtt{t_1}$ und $\mathtt{x_2} : \mathtt{t_2}$ erweiterten Umgebung ergeben muss. In einem Ausdruck kann der geschachtelte λ-Ausdruck nun wahlweise mit null, ein oder zwei Argumenten versorgt vorkommen. Eine (nur scheinbare) *Überversorgung* ist möglich, wenn der Typ `t` des Rumpfes `e` ein Funktionstyp ist, d.h. auch der Ergebnistyp `t` bestimmt die maximale Stelligkeit des λ-Ausdrucks.

Unterversorgte Funktionsapplikationen oder scheinbar überversorgte λ-Ausdrücke beeinträchtigen also in erster Linie nur die Lesbarkeit für (unbedarfte) Benutzer. Überlagerungsauflösung, ohnehin nur eine endliche Fallunterscheidung, ist nach wie vor möglich. Weil die Stelligkeit von *curried* Funktionen nicht genau feststeht, sind potenzielle Mehrdeutigkeiten, die durch Annotationen vermieden werden müssen, etwas wahrscheinlicher.

3.4 Verschattung und Überlagerung

Eine Zusatzkomplikation bei der Überlagerungsauflösung ergibt sich, wenn man *lokale* Variablen wie z.B. λ-Variablen oder *formale* Argumente $\mathtt{x_i}$ von definierenden Funktionsgleichungen betrachtet. In der Praxis wird dieser Fall ausgeschlossen, indem ein lokaler Bezeichner einfach *alle* überlagerten *globalen* Namen *verschattet* und dadurch eindeutig wird.

Ein Verbot der Überlagerung für lokale Variablen würde später deklarierte globale Bezeichner einschränken bzw. nachträgliche Umbenennungen erfordern. Eine lokale Verschattung dagegen wirkt sich nicht auf andere globale Namen und definierende Ausdrücke aus. Im lokalen Kontext sind dafür allerdings verschattete globale Namen unbekannt.

Gibt es z.B. eine globale *Funktion* `x`, dann kann diese bei Verschattung durch eine lokale λ-*Konstante* in $\lambda\mathtt{x}:\mathtt{t}.\mathtt{e}$ im Rumpfausdruck `e` nicht appliziert werden. Eine Applikation der Funktion `x` im Ausdruck `e` wäre dann typfalsch, während derselbe Ausdruck `e` mit einer global vereinbarten Konstante $\mathtt{x}:\mathtt{t}$ durch die Berücksichtigung der Überlagerung typisierbar sein könnte.

Für einen orthogonalen Sprachentwurf ist es wünschenswert, dass lokale und globale Namen weitgehend gleich behandelt werden. Wenn der Typ für ein lokales Argument $\mathtt{x}:\mathtt{t}$ explizit gegeben ist, bräuchte eigentlich nur das *globale* `x` überschrieben werden, das vom selben Typ `t` ist; alle anderen (globalen) `x`-Einträge stünden dann für die Überlagerungsauflösung zur Verfügung. Der λ-Ausdruck $\lambda\mathtt{x}:\mathtt{t}.\mathtt{x}(\mathtt{x})$ könnte z.B. aufgelöst werden, wenn eine globale Funktion `x` mit passendem Typ vorliegt.

Leider können durch Überlagerungen mit lokalen Variablen unerwünschte Mehrdeutigkeiten entstehen. Seien beispielsweise globale, überlagerte Funktionen `f` mit den Typen $\mathtt{t_1}\to\mathtt{t}$ und $\mathtt{t_2}\to\mathtt{t}$ sowie eine globale Konstante $\mathtt{x}:\mathtt{t_1}$ gegeben: betrachtet man den Ausdruck $\lambda\mathtt{x}:\mathtt{t_2}.\mathtt{f}(\mathtt{x})$, dann ist der Rumpf $\mathtt{f}(\mathtt{x})$ auf Grund der Überlagerungen von `x` und `f` *mehrdeutig*. Intuitiv sollte aber die *lokale* Variable $\mathtt{x}:\mathtt{t_2}$ *vorrangig* die Auflösung der Überlagerung von $\mathtt{f}$ zu $\mathtt{f}:\mathtt{t_2}\to\mathtt{t}$ bewirken. Im Prinzip sollte der bei einer Verschattung typkorrekte Ausdruck $\lambda\mathtt{x}:\mathtt{t_2}.\mathtt{f}(\mathtt{x})$ *trotz Überlagerung* durch das globale $\mathtt{x}:\mathtt{t_1}$ typkorrekt bleiben. Eine einfache Doppelanalyse, erst Überlagerungsauflösung dann Analyse mit Verschattung, ist unbefriedigend, wenn z.B. im obigen Beispiel `f` durch `x` ersetzt wird:

$$\mathtt{x}:\{\mathtt{t_1},\mathtt{t_1}\to\mathtt{t},\mathtt{t_2}\to\mathtt{t}\}$$

Der Ausdruck $\lambda \mathtt{x} : \mathtt{t}_2.\mathtt{x}(\mathtt{x})$ ist bei Überlagerung mehrdeutig und bei Verschattung typfalsch. Eine Bevorzugung von lokalen Variablen bei der Überlagerungsauflösung, um den Verschattungsaspekt zu integrieren, ist nicht leicht zu realisieren, obwohl das Problem *endlich* ist. Dazu betrachte man den Ausdruck $\lambda \mathtt{x} : \mathtt{t}_2.\mathtt{h}(\mathtt{x}, \mathtt{x})$ mit der folgenden globalen Umgebung:

$$\mathtt{x} : \mathtt{t}_1$$
$$\mathtt{h} : \{\mathtt{t}_1 \times \mathtt{t}_2 \to \mathtt{t}, \mathtt{t}_2 \times \mathtt{t}_1 \to \mathtt{t}\}$$

Für die durch `h` typverschiedenen beiden Argumente `x` kann das lokale $\mathtt{x} : \mathtt{t}_2$ nur in einem Fall vor dem globalen $\mathtt{x} : \mathtt{t}_1$ bevorzugt werden. Beide Auflösungen sind symmetrisch und deswegen sollte der Gesamtausdruck mehrdeutig sein. Eine Bevorzugung der ersten Komponente (wie in PVS) ist Willkür. Eine Mehrheitsentscheidung bei weiteren Komponenten oder eine Wichtung für mehr oder weniger lokale Variablen aus verschachtelten λ-Ausdrücken wäre nicht mehr nachvollziehbar.

Durch eine zusätzliche Überlagerung von `h` mit Typ $\mathtt{t}_2 \times \mathtt{t}_2 \to \mathtt{t}$ wäre der Ausdruck $\lambda \mathtt{x} : \mathtt{t}_2.\mathtt{h}(\mathtt{x}, \mathtt{x})$ durch die einheitliche Bevorzugung von $\mathtt{x} : \mathtt{t}_2$ wieder eindeutig auflösbar. Die Auflösbarkeit ist dann aber nicht *monoton* (siehe Abschnitt 6.4), da ein größerer Namensraum nicht mehr, sondern sogar weniger Mehrdeutigkeiten verursachen kann.

Im Kontext von Funktionen höherer Ordnung können natürlich auch lokale λ-Variablen vom Funktionstyp sein: der Ausdruck $\lambda \mathtt{x} : \mathtt{t}_1 \to \mathtt{t}_1.\mathtt{x}(\mathtt{x})$ sollte eigentlich eindeutig auflösbar sein, wenn $\mathtt{x} : \mathtt{t}_1$ global bekannt ist und zwar unabhängig von beliebigen weiteren globalen Überlagerungen für `x`, da die lokale Funktion $\mathtt{x} : \mathtt{t}_1 \to \mathtt{t}_1$ in jedem Fall *bevorzugt* zu applizieren ist. Im Prinzip steht man also bei diesem kleinen Teilaspekt des Sprachentwurfs vor einem Dilemma:

- Die vollständige Verschattung durch Überschreiben aller überlagerten Namen ist ziemlich restriktiv und unintuitiv in einigen Fehlerfällen. Manchmal wird der Namensraum sogar kleiner.
- Die weitgehende Gleichberechtigung von lokalen und globalen Namen, indem nur *typgleiche* überlagerte Namen überschrieben werden, kann zu unintuitiven Mehrdeutigkeiten führen, weil lokale Namen nicht bevorzugt behandelt werden.
- Eine Überlagerungsauflösung mit einer Bevorzugung für lokale Namen vereinigt zwar die Vorteile von Verschattung und Überlagerung, ist aber

kompliziert und konfrontiert den Benutzer mit einer zusätzlichen Variante von Mehrdeutigkeitsfehlern. Außerdem geht eine (später relevante) *Monotonieeigenschaft* für Namensräume verloren.

Statt der üblichen Entwurfsentscheidung für Verschattung wird hier konsequente Überlagerungsauflösung bevorzugt:

- Die Namenswahl ist weniger restriktiv und eine *Gleichbehandlung* von lokalen und globalen Namen am ehesten möglich.
- Die Erweiterung des Namensraums durch lokale Variablen ist in gewisser Weise *monoton*.
- Die unerwünschten Mehrdeutigkeitsfehler müssen bei Bedarf durch *Annotationen* beseitigt werden. Für λ-Variablen ist dabei auch eine besondere Herkunftsannotation – etwa de-Bruijn-Indizes oder die Verschachtelungstiefe – vorstellbar.
- Im Hinblick auf eine zusätzliche Bevorzugung für lokale Namen ist die vorherige Kenntnis *aller* Überlagerungsmöglichkeiten zumindest ein guter Ausgangspunkt.

Kapitel 4

Algebraische Typanalyse

Für die algebraische Typanalyse werden nun die polymorphe Typinferenz (Kapitel 2) und die Überlagerungsauflösung (Kapitel 3) verschmolzen. Der Ausgangspunkt sind die elementaren Funktionsausdrücke `expr` (von Core-ML), die die für die Überlagerungsauflösung betrachteten einfachen Applikationsausdrücke `appl`[`ide`] umfassen. Algebraisch wird eine Umgebung mit überlagerten und polymorphen Funktionen durch Importe mehrerer Strukturen sowie Funktionsdeklarationen etabliert. Dieser Aspekt der Signaturanalyse wird in Kapitel 5 ausgeführt. Es wird hier von einer globalen Umgebung ausgegangen, die auch durch initiale polymorphe LET-Ausdrücke entstehen kann; nur Funktionen sind relevant, Typen (und Typkonstruktoren) dagegen müssen wie in ML eindeutig (und korrekt appliziert) sein. Insofern ist die algebraische Typanalyse nichts anderes als eine Erweiterung der ML-Polymorphie um Überlagerung. Die Parametrisierung mit Funktionen wird als ein Teil der Signaturanalyse behandelt.

Auf Grund der schon für monomorphe Ausdrücke diskutierten Probleme muss man auf *Verschattung* (Abschnitt 3.4) verzichten. Auch ohne Überlagerung ist Verschattung problematisch, da beispielsweise das Entfernen einer lokalen Bindung die Semantik eines Ausdrucks ändern und bestenfalls einen Typfehler verursachen kann. Die Verschattung eines typgleichen Namens bei der Überlagerungsauflösung für λ-Ausdrücke mit *explizit typisierten λ-Variablen* ist zwar leicht zu implementieren, aber methodisch fragwürdig. Bei polymorphen Typen können möglicherweise nur spezielle Instanzen gleich werden:

$\mathtt{f} : \alpha \times \mathtt{nat} \to \alpha$
$\mathtt{f} : \mathtt{nat} \times \alpha \to \alpha$

Für die Instanz $[\alpha := \texttt{nat}]$ sind beide Funktionen `f` ununterscheidbar, z.B. in einer Applikation $\texttt{f}(\texttt{n},\texttt{n})$, wenn `n` vom Typ `nat` ist. Selbst eine *monomorphe* Typannotation zur Funktion $\texttt{f} : \texttt{nat} \times \texttt{nat} \to \texttt{nat}$ ist keine Unterscheidungshilfe.

Möchte man eine derartige Konstellation nicht verbieten (vgl. Abschnitt 5.4.2), dann ist es durchaus konsequent, überlagerte Funktionen zu verwalten, die denselben polymorphen oder auch denselben monomorphen Typ aufweisen.

Typgleiche überlagerte Funktionen führen zu Mehrdeutigkeiten; es ist aber ausreichend, solche Fehler erst für konkrete Applikationen zu melden, wenn sie dann durch (Herkunfts-) Annotationen problemlos zu korrigieren sind. Ebenso kann eine Funktion nie ohne Mehrdeutigkeit appliziert werden, wenn ihr Typschema *spezieller* als das einer Überlagerung ist. Eine einfache, aber unsymmetrische Realisierung der Verschattung durch eine *lokale* Funktion beträfe nur *speziellere* überlagerte globale Funktionen:

$$\textsc{let}\ \texttt{x} = \texttt{1}\ \textsc{in}\ \textsc{let}\ \texttt{y} = \lambda\texttt{x.x}\ \textsc{in}\ \ldots$$

Der innere, lokale Bezeichner $\texttt{x} : \alpha$ sollte das äußere, globale $\texttt{x} : \texttt{nat}$ bei der Analyse des λ-Ausdrucks $\lambda\texttt{x.x}$ eigentlich verschatten. Im folgenden λ-Ausdruck $\lambda\texttt{x.x}(\texttt{x})$ dagegen darf das globale $\texttt{x} : \texttt{nat}$ nicht verschattet werden, um als Typ für den lokalen Bezeichner $\texttt{x} : \texttt{nat} \to \alpha$ inferieren zu können:

$$\textsc{let}\ \texttt{x} = \texttt{1}\ \textsc{in}\ \textsc{let}\ \texttt{y} = \lambda\texttt{x.x}(\texttt{x})\ \textsc{in}\ \ldots$$

Wären die Typen der λ-Variablen explizit gegeben, dann könnte man durch Typvergleich Verschattung oder Überlagerung wählen. Da die λ-Variablen aber *ungetypt* sind, muss man sich *vorher* entscheiden und wird dann die i.A. nützlichere *Überlagerung* favorisieren.

Ohne Verschattung wird die bekannte Identitätsfunktion $\lambda\texttt{x.x}$ genau dann mehrdeutig, wenn `x` schon in der globalen Umgebung vorkommt. Das ist nicht schlechter als ein Verbot der Überlagerung durch λ-Variablen, denn dafür wäre $\lambda\texttt{x.x}$ ebenfalls nur korrekt, wenn `x` noch nicht global bekannt ist.

Um bei der Überlagerungsauflösung zumindest lästige Konflikte mit beliebig typisierten *unbenutzten* lokalen Variablen zu vermeiden, kann in Opal und anderen Sprachen der Unterstrich in Lambda- ($\lambda\ \_\ .\texttt{e}$) oder musterbasierten let-Ausdrücken verwendet werden.

4.1 Analysealgorithmus $\mathcal{W}_o$

Für die betrachteten Ausdrücke `expr` wird von der (aus Kapitel 2) bekannten Typableitungsrelation $\mathtt{A} \vdash \mathtt{e} : \mathtt{t}$ ausgegangen. Die Regeln bleiben unverändert; nur die Umgebung `A` wird flexibler. Die (VAR)-Regel kann für alle möglichen Überlagerungen angewendet werden; λ- und LET-Variablen werden einfach *additiv* zur Umgebung hinzugefügt. Die Umgebung $\mathtt{A} : \mathtt{env}$ kann man sich also als Liste organisiert vorstellen.

Analog zu Algorithmus $\mathcal{W}$ lässt sich der folgende bottom-up Algorithmus $\mathcal{W}_o$ angeben. (Der in [Smi91] ebenso benannte Algorithmus ist anders implementiert und wird in Abschnitt 4.3 diskutiert.) Vergleichbar mit der bruteforce Überlagerungsauflösung (Kapitel 3) werden hier alle möglichen korrekten Typisierungen für einen Ausdruck berechnet. Das Ergebnis ist eine *endliche* Liste von prinzipalen Typen mit zugehörigen Substitutionen:

$$\text{FUN } \mathtt{W_o} : \mathtt{env} \times \mathtt{expr} \times \mathtt{nat} \to \mathtt{seq}[\mathtt{type} \times \mathtt{subst} \times \mathtt{nat}]$$

Ohne Überlagerung in der Umgebung berechnet $\mathcal{W}_o$ dasselbe wie $\mathcal{W}$, nur ist dann das Ergebnis eine maximal einelementige Liste vom Typ `seq` statt vom Typ `option`. Insofern ist $\mathcal{W}_o$ eine echte Verallgemeinerung von $\mathcal{W}$. Mehrelementige Listen durch Überlagerungen signalisieren Mehrdeutigkeiten; für mehrdeutige Zwischenergebnisse werden in den Abschnitten 4.2 und 4.3 alternative Behandlungsmöglichkeiten diskutiert. Der – bis auf den Elementtest unveränderte – spezifizierende Zusammenhang zwischen $\mathcal{W}_o$ und den Ableitungsregeln wird wieder durch die Konsistenz und Vollständigkeit (für alle $\mathtt{S}, \mathtt{A}, \mathtt{e}, \mathtt{t}$) angegeben:

$$(\mathtt{t}, \mathtt{S}, \_) \in \mathtt{W_o}(\mathtt{A}, \mathtt{e}, \_) \;\Rightarrow\; \mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{t}$$
$$\mathtt{S}(\mathtt{A}) \vdash \mathtt{e} : \mathtt{t} \;\Rightarrow\; \exists \mathtt{t_0}\ \mathtt{R}.\ (\mathtt{t_0}, \mathtt{S}, \_) \in \mathtt{W_o}(\mathtt{A}, \mathtt{e}, \_) \;\wedge\; \mathtt{t} = \mathtt{R}(\mathtt{t_0})$$

Algorithmus $\mathcal{W}_o$ entsteht aus $\mathcal{W}$ mit den schon für die Überlagerungsauflösung benutzten *Listenkomprehensionen*; eine vormals monadische Komposition $\mathtt{V} := \mathtt{S}; \mathtt{E}$ über dem Typ `option` wird nun für Listen `seq` eine Komprehension $\{\mathtt{E} \mid \mathtt{V} \in \mathtt{S}\}$, mit dem Elementtest $\mathtt{V} \in \mathtt{S}$ als *Generator*. Mehrfache ;-Kompositionen in $\mathcal{W}$ werden in $\mathcal{W}_o$ zu einer Verknüpfung mit dem logischen Und ($\wedge$). Eine Unifikation im Bedingungsteil einer Komprehension muss natürlich erfolgreich sein, um zum Ergebnis beizutragen. Die Struktur und Bezeichner sind ansonsten identisch zum Algorithmus $\mathcal{W}$:

$$\mathtt{W_o(A,x,n)} =$$
$$\quad \{(\mathtt{inst(n,A(x))}, \epsilon, \mathtt{n} + \#(\mathtt{bv(A(x))})) \mid \mathtt{x} \in \mathtt{A}\}$$

$$\mathtt{W_o(A,f(e),n)} =$$
$$\quad \{(\mathtt{S_3(l)}, \mathtt{S_3} \circ \mathtt{S_2} \circ \mathtt{S_1}, \mathtt{l}+1) \mid$$
$$\quad (\mathtt{t}, \mathtt{S_1}, \mathtt{m}) \in \mathtt{W_o(A,f,n)} \wedge$$
$$\quad (\mathtt{t_1}, \mathtt{S_2}, \mathtt{l}) \in \mathtt{W_o(S_1(A),e,m)} \wedge$$
$$\quad \mathtt{S_3} = \mathtt{unify(S_2(t), t_1 \to tvar(l))}\}$$

$$\mathtt{W_o(A,\lambda x.e,n)} =$$
$$\quad \{(\mathtt{S(n)} \to \mathtt{t}, \mathtt{S}, \mathtt{m}) \mid$$
$$\quad (\mathtt{t}, \mathtt{S}, \mathtt{m}) \in \mathtt{W_o(A + x : tvar(n), e, n+1)}\}$$

$$\mathtt{W_o(A,LET\ x_1 = e_1 IN\ e,n)} =$$
$$\quad \{(\mathtt{t}, \mathtt{S_2} \circ \mathtt{S_1}, \mathtt{l}) \mid$$
$$\quad (\mathtt{t_1}, \mathtt{S_1}, \mathtt{m}) \in \mathtt{W_o(A, e_1, n)} \wedge$$
$$\quad (\mathtt{t}, \mathtt{S_2}, \mathtt{l}) \in \mathtt{W_o(S_1(A) + x_1 : gen(S_1(A), t_1), e, m)}\}$$

$$\mathtt{W_o(A,FIX\ x.e,n)} =$$
$$\quad \{(\mathtt{S_2(t)}, \mathtt{S_2} \circ \mathtt{S_1}, \mathtt{m}) \mid$$
$$\quad (\mathtt{t}, \mathtt{S_1}, \mathtt{m}) \in \mathtt{W_o(A + x : tvar(n), e, n+1)} \wedge$$
$$\quad \mathtt{S_2} = \mathtt{unify(S_1(n), t)}\}$$

Zusammen mit dem Typ und der Substitution könnte auch der *aufgelöste Ausdruck* synthetisiert werden. Die Bezeichner in einem aufgelösten Ausdruck würden dabei eindeutig *annotiert*, z.B. mit ihren Positionen in der Umgebungsliste.

Die Korrektheit des Algorithmus $\mathcal{W}_o$, also die Konsistenz mit der Spezifikation, entspricht der von Algorithmus $\mathcal{W}$ mit Fallunterscheidungen für alle Überlagerungen. Die angegebene bottom-up Variante ist am leichtesten verständlich. (Der Zähler `n` könnte auch global und scheinbar imperativ in einer Zustandsmonade verwaltet werden, wenn man jeweils mit dem *Maximum* aus einer Liste weiter rechnet.)

Später wird für die Namensidentifikation $\mathcal{I}$ (Kapitel 6) eine analoge top-down Variante basierend auf Algorithmus $\mathcal{M}$ und der (brute-force) Cormack-Überlagerungsauflösung angegeben. Eine Integration der polynomialen Überlagerungsauflösung mit Attributen (Abschnitt 3.2) fällt schon schwerer, insgesamt kann sich allein durch die LET-Polymorphie kein polynomialer Algorithmus ergeben.

Offen ist allerdings, wie für eine gegebene Umgebung und *monomorphe* LET-Ausdrücke, die typisch algebraisch sind, die Typanalyse insgesamt polynomial implementiert werden kann. Notwendige Voraussetzungen dafür sind sicherlich eine polynomiale Unifikation und die Graphrepräsentation von Typtermen. Allein bei Applikationen muss eine Explosion durch Tupel vermieden werden; das naive Durchprobieren von $\mathtt{o}$ Überlagerungen für jede von $\mathtt{k}$ Komponenten – also von $\mathtt{o}^{\mathtt{k}}$ Kombinationen – wäre unakzeptabel. Wie beim top-down Algorithmus $\mathcal{I}$ zur Namensidentifikation (Kapitel 6) müssen Spezialisierungen von Typvariablen einer Komponente direkt in die Analyse abhängiger Komponenten (mit denselben Typvariablen) einfließen. Dadurch werden effizienter *Schnittmengen* für Spezialisierungen von mehrfach vorkommenden Typvariablen gebildet. Zusätzlich müssten für unabhängige Komponenten frühzeitig Mehrdeutigkeiten signalisiert werden. Ohne konkrete Implementierung kann man hier die Existenz eines polynomialen Algorithmus nur vermuten.

Eine Kernfrage zur Typanalyse ist aber durch die zusätzlichen Überlagerungen nicht nur die Korrektheit von $\mathcal{W}_o$, sondern die Validität der Typableitungsregeln bezüglich der erweiterten Umgebung. Ist diese Spezifikation also das, was man haben möchte? In den folgenden beiden Abschnitten werden dazu Vor- und Nachteile von alternativen Interpretationsmöglichkeiten für Mehrdeutigkeiten (am LET) erläutert.

4.2 Restriktion: Lokale LET-Eindeutigkeit

Die Verbindung von Typinferenz und Überlagerung ist in folgender Hinsicht erklärungsbedürftig: im obigen Algorithmus hängt der Typ für eine LET-Variable nicht nur vom definierenden Ausdruck ab, sondern auch von der *Benutzung* (Applikation) der Variablen im nachfolgenden IN-Ausdruck. Das folgende LET $\mathtt{z} = \mathtt{x}$ wird z.B. mit den Überlagerungen für $\mathtt{x}$ erst durch die spätere Applikation $\mathtt{z} + \mathtt{1}$ (nach dem letzten IN) eindeutig auflösbar:

```
LET x = 1 IN
LET x = λy.y IN     - überlagert!
LET z = x IN        - mehrdeutig?
z + 1
```

Überlagerung ist ein globaler Aspekt, der sich an dieser Stelle nicht gut mit einer inkrementellen, modularen Sichtweise verträgt. Einige LET-Ausdrücke würde man gerne als Bibliotheksfunktionen betrachten, deren (polymorphe)

Typen auch lokal, ohne Kenntnis einer zukünftigen Verwendung, eindeutig inferiert werden sollen.

Eine entsprechende Modifikation im Algorithmus würde also einfach die eindeutige Typisierbarkeit für den definierenden Teilausdruck $\mathtt{e_1}$ voraussetzen:

$$
\begin{array}{l}
\mathtt{W_o(A,}\textsc{let}\ \mathtt{x_1 = e_1}\textsc{in}\ \mathtt{e, n)} = \\
\quad \textsc{if}\ \mathtt{\#(W_o(A, e_1, n))} = \mathtt{1}\ \textsc{then} \\
\quad\quad \mathtt{\{(t, S_2 \circ S_1, l)} \mid \\
\quad\quad \mathtt{(t_1, S_1, m) = ft(W_o(A, e_1, n))} \wedge \\
\quad\quad \mathtt{(t, S_2, l) \in W_o(S_1(A) + x_1 : gen(S_1(A), t_1), e, m)\}} \\
\quad \textsc{else}\ \Diamond\ \textsc{fi} \qquad\qquad \text{-}\textit{Typfehler}
\end{array}
$$

Für die zugehörige (LET)-Typableitungsregel könnte dann allerdings eine Anwendungsvoraussetzung $\mathtt{unique(A, e_1)}$ nicht leicht ergänzt werden. Umgebungen in den Typableitungsregeln können speziellere Typannahmen enthalten als die berechneten Umgebungen im Algorithmus:

$$
\begin{array}{l}
\lambda\mathtt{x}. \\
\quad \textsc{let}\ \mathtt{x = 1}\ \textsc{in} \\
\quad \textsc{let}\ \mathtt{y = x + 1}\ \textsc{in} \\
\quad \mathtt{x(1) + y}
\end{array}
$$

Der Ausdruck $\mathtt{x + 1}$ ist in der *berechneten* Umgebung $\{\mathtt{x}: \alpha, \mathtt{x}: \mathtt{nat}\}$ *mehrdeutig*, aber eindeutig in der (für $\mathtt{x(1) + y}$ richtig) spezialisierten Umgebung $\{\mathtt{x}: \mathtt{nat} \to \mathtt{nat}, \mathtt{x}: \mathtt{nat}\}$, die in einer Typableitung gewählt werden könnte.

Dieses Problem besteht nicht, wenn $\mathtt{e_1}$ (aus $\mathtt{unique(A, e_1)}$) keine Variablen mit unbekannten Typen in $\mathtt{A}$ enthält, also insbesondere dann, wenn die ganze Umgebung $\mathtt{A}$ keine Unbekannten enthält. Dieses ist z.B. für PVS mit obligatorisch und eindeutig getypten λ-Variablen der Fall.

Die lokale LET-Eindeutigkeit ist insofern restriktiv. Ohne Restriktion dagegen ist mit der Syntax von Core-ML keine modulare, sondern nur eine einzige globale Überlagerungsauflösung möglich, in die sämtliche Applikationsstellen von Variablen einfließen. Ein Kompromiss wäre, lokale Eindeutigkeit nur für äußere LET-Ausdrücke zu fordern, die eine initiale und fest typisierte globale Umgebung bilden sollen. Nur für LET-Variablen innerhalb von λ-Ausdrücken würden dann Applikationsstellen berücksichtigt.

In OPAL ist dieser Kompromiss für die *deklarierten* Funktionen einer Struktur realisiert. Der Typ dieser Funktionen muss eindeutig vorliegen und für einen *definierenden* Ausdruck wird Überlagerung gemäß $\mathcal{W}_o$ aufgelöst. Die

Lösung für *monomorphe* LET-Ausdrücke entspricht dabei der von äquivalenten λ-Ausdrücken. (Mehrfach-)Vorkommen einer λ-Variablen mit einem festen Typ entsprechen den Applikationsstellen von LET-Variablen.

Denkbar ist es auch, Überlagerungen zusammen für alle Ausdrücke einer Struktur aufzulösen. Je größer allerdings der zur Typauflösung tatsächlich benötigte Kontext ist, desto schwerer verständlich wird auch der Quelltext. Deswegen sollten Funktionsrümpfe und (ungetypte) λ-Ausdrücke nicht „zu groß“ werden.

Entwurfstheoretisch und implementierungstechnisch *einfach* wäre es, wie bei der klassischen Überlagerungsauflösung, auf λ-Ausdrücke innerhalb definierender Gleichungen zu verzichten. Monomorphe LET-Ausdrücke können beibehalten werden. Dadurch wird die *gesamte* Kontextanalyse *lokal eindeutig* und *monomorph* (Kapitel 9). In konkreten OPAL-Quellen werden λ-Ausdrücke insbesondere für monadische Kompositionen zur Ein-/Ausgabe extensiv benutzt. *Ohne λ-Ausdrücke* wird also in der Praxis die Kompatibilität und Akzeptanz relevant.

4.3 Verallgemeinerung: Überlagertes LET

Alternativ könnten einer LET-Variablen auch *mehrere* Typisierungen zugeordnet werden. In der Typableitungsregel für LET würde die Umgebung nicht nur um *einen* Eintrag für die LET-Variable $\mathtt{x}$, sondern um eine beliebige, aber endliche Liste $\mathtt{T_1}$ von überlagerten Einträgen erweitert:

$$\frac{\mathtt{t_1} \in \mathtt{T_1} \Rightarrow \mathtt{A} \vdash \mathtt{e_1} : \mathtt{t_1} \qquad (\mathtt{A} + \{\mathtt{x_1} : \mathtt{gen}(\mathtt{A}, \mathtt{t_1}) \mid \mathtt{t_1} \in \mathtt{T_1}\}) \vdash \mathtt{e} : \mathtt{t}}{\mathtt{A} \vdash (\text{LET } \mathtt{x_1} = \mathtt{e_1} \text{ IN } \mathtt{e}) : \mathtt{t}}$$

Diese Regel hätte den Vorteil, dass man sich den LET-Ausdruck als Substitution $\mathtt{e}[\mathtt{x_1} := \mathtt{e_1}]$ für die Vorkommen der LET-Variablen $\mathtt{x_1}$ vorstellen kann. Der Ausdruck $\mathtt{e_1}$ kann also in $\mathtt{e}$ an unterschiedlichen Stellen nicht nur unterschiedlich instanziiert, sondern auch unterschiedlich überlagert vorkommen. Eine Implementierung dieser Typableitungsregel bereitet aber Probleme:

- Die Substitution $\mathtt{e}[\mathtt{x_1} := \mathtt{e_1}]$ kann wegen einer potenziellen Überlagerung von $\mathtt{x_1}$ syntaktisch nicht durchgeführt werden. Die Typanalyse von $\mathtt{e_1}$ muss also vorher und unabhängig von $\mathtt{e}$ erfolgen. Gemäß der Überlagerungen in $\mathtt{e_1}$ muss zudem $\mathtt{x_1}$ typabhängiger Code zugeordnet werden.

- Betrachtet man die rekursiv berechnete Liste $\mathtt{W_o(A, e_1, n)}$ als mögliche Typen für $\mathtt{x_1}$, dann könnten die zugehörigen Substitutionen für die Umgebung verschieden und sogar inkompatibel sein. Die Typanalyse von $\mathtt{e_1}$ hängt in diesem Fall z.B. von übergeordneten λ-Variablen ab, deren Typen noch unbekannt sind und sich erst aus der Typanalyse von $\mathtt{e_1}$ (und von $\mathtt{e}$) ergeben können.

In der Literatur [Smi91, CF99] zur Verschmelzung von Überlagerung mit Polymorphie – in [Smi91] heißt der Algorithmus auch $\mathcal{W}_o$ – werden üblicherweise die Typen von überlagerten Funktionen zu einem einzigen polymorphen Typ zusammengefasst, nämlich der *kleinsten gemeinsamen Verallgemeinerung* (lcg, *least common generalisation*), die durch Anti-Unifikation berechnet wird. In einigen Fällen entsteht dabei der nichtssagende Typ $\forall\alpha.\alpha$, der nur in Verbindung mit Beschränkungen (*Constraints*) Sinn macht, deren Erfüllbarkeit dann zum eigentlichen Auflösungsproblem wird. Durch die Zusammenfassung der Überlagerungen wird ansonsten nur die übliche polymorphe Typinferenz gemäß Algorithmus $\mathcal{W}$ benötigt. Die betrachteten Funktionen und LET-Ausdrücke sind aber nur zum Zwecke der Typinferenz *parametrisch* polymorph, semantisch aber keineswegs *uniform*. Ein solches Typsystem – in [CF99] CT genannt – ist mit HASKELL und *impliziten* Typklassen vergleichbar.

4.4 Monomorphie

Eine typische Beschränkung für algebraische Sprachen sind nicht nur monomorphe LET-Ausdrücke, sondern insgesamt eine im Prinzip monomorphe Typanalyse, weil explizite Typparameter (statt impliziter Typvariablen) verwendet werden und sich für generische Funktionen (aus generischen Importen) letztendlich monomorphe Instanzen ergeben müssen. Der Typ eines Ausdrucks muss also nicht nur eindeutig, sondern auch noch *variablenfrei* sein. Insbesondere müssen sogar die Typen für alle Teilausdrücke variablenfrei sein. Diese Beschränkung gilt für die ML-Polymorphie nicht.

Der Ausdruck $\mathtt{\#}(\lozenge)$, also die Länge der leeren Liste, ist in ML (als $\mathtt{len(nil)}$) korrekt typisiert, auch ohne Kenntnis der konkreten Listeninstanz. Die operationale Berechnung ist *uniform* und tatsächlich unabhängig von der Instanz. Bei Bedarf könnte sogar irgendeine Instanz gewählt werden. Konsequenterweise könnte man auch für *überlagerte* Instanzen *keine* Mehrdeutigkeit für $\mathtt{\#}(\lozenge)$ diagnostizieren.

Der Ergebnistyp von `#`($\Diamond$) ist monomorph `nat`, das alleine reicht (in PVS und OPAL) nicht aus. Die algebraisch geforderte strenge Monomorphie ergibt sich aus einer konsequenten bzw. übertriebenen Gleichbehandlung von Typen und Funktionen. Bei einer Parametrisierung mit Funktionen (Kapitel 5) ist natürlich die Kenntnis der Instanz für die operationale Berechnung unverzichtbar.

Im Prinzip ist es eine eher unbedeutende Entwurfsentscheidung, ob man freie Typvariablen in Teilausdrücken akzeptiert oder verbietet. Ein Monomorphietest ist auf jeden Fall nötig, wenn Überlagerungen wie in [CF99] (und Abschnitt 4.3) zu einer nur scheinbar uniformen Funktion zusammengefasst werden. Die Semantik eines Ausdrucks wie `#`($\Diamond$) – oder typischer `show` $\circ$ `read` – ist dann ggf. doch von der Instanz abhängig. Und auch in HASKELL gibt es die so genannte *Monomorphie-Restriktion*, um die Eindeutigkeit einer Typklasseninstanz (für `show`) zu garantieren.

Eine `show`-Funktion für Listen wäre algebraisch mit einer ggf. anders benannten `show`-Funktion für den Elementtyp parametrisiert. Durch diese Parametrisierung ist die ansonsten *uniforme* Behandlung für Listen explizit durch die Instanz (wie durch ein Argument höherer Ordnung) *am Typ* erkennbar.

Insgesamt wird durch den Algorithmus $\mathcal{W}_o$ eine enge und adäquate Verschmelzung von parametrischer und Ad-hoc-Polymorphie erzielt. Die Überlagerung ist flexibel und einzelne (LET-) Namen bleiben *uniform*. Außerdem brauchen generische Namen nicht a priori auf endlich viele monomorphe Instanzen beschränkt werden.

Kapitel 5

Namen und Instanzen

In den Typanalysealgorithmen der vorherigen Kapitel wurden explizite Typsignaturen ignoriert, weil ihre Integration trivial war. Alle Typbezeichner mussten global eindeutig sein und lediglich gemäß ihrer Stelligkeit mit der korrekten Anzahl von Typargumenten versorgt worden sein.

Im algebraischen Kontext ergibt sich aus der Parametrisierung mit Funktionen und der Gleichberechtigung von Typen und Funktionen, dass nicht nur Funktionen sondern auch Typen überlagert sein können. Dazu betrachten wir im folgenden Abschnitt 5.1 die Struktur `Set` für endliche Mengen.

Der Funktionsparameter *enthält* den Typparameter für Elemente der Mengen; diese *Abhängigkeit* für eine *Instanziierung* kann in Analogie zur *Applikation* von überlagerten und polymorphen Funktionen behandelt werden. In Abschnitt 5.2 wird dafür zunächst die Instanziierung auf die klassische *Substitution* von speziellen *Variablen* zurückgeführt. Eine wichtige Erkenntnis dabei ist, dass nur der vordere Teil eines vollständigen Namens substituiert wird. Variablen stehen also – unabhängig davon, ob es sich um einen Typ oder eine Funktion handelt – für eine *Namensfront* oder *Namensinstanz*. (Beide Begriffe sind auf die eine oder andere Weise plausibel, aber noch gewöhnungsbedürftig.)

Die *vollständigen* Namen und *Namensinstanzen* `inst` – eine Verallgemeinerung der klassischen Typterme mit Typvariablen – werden in Abschnitt 5.3 formalisiert und alternative Modellierungen erläutert. Die *Unifikation* formaler und aktueller Parameterlisten liefert Instanzen für *generische* Namen.

Den vollständigen Namen werden in Abschnitt 5.4 die *partiellen* Namen gegenübergestellt. Die partiellen Namen sind die syntaktischen Vorgaben des

Quelltextes und letztendlich ist es die Aufgabe der *Namensidentifikation* aus Kapitel 6 (in Analogie zur polymorphen Typanalyse mit Überlagerung aus Kapitel 4) herauszufinden, für welchen vollständigen Namen ein partieller Name eigentlich steht. In Abschnitt 5.5 werden Funktionen höherer Ordnung durch die Parametrisierung mit Funktionen diskutiert.

5.1 Endliche Mengen

Die folgende Struktur `Set` für endliche Mengen, bzw. nur ihr Signaturteil, ist charakteristisch und herausragend für eine algebraische Sprache wie OPAL:

```
STRUCTURE Set[α, <]
  TYPE α                          - Parameter
  FUN <: α × α → bool

  TYPE set                        - Datentyp
  FUN {} : set                    - Konstruktor
  FUN incl : α × set → set
  FUN {}? : set → bool            - Test
  FUN in : α × set → bool
  FUN min : set → α               - Selektor
      ...
  FUN <: set × set → bool         - totale Ordnung
      ...
  IMPORT Seq[α] COMPLETELY        - Implementierung
```

Die Struktur `Set` ist wie die Struktur `SeqOrd` (Abschnitt 1.3.1) mit einer totalen Ordnung parametrisiert. `Set` enthält einen *Typ* `set` und zugehörige Bearbeitungsfunktionen. (Zur fehlenden Implementierung siehe Abschnitt 8.4.4). In HASKELL sähen vergleichbare Mengen mit einem *Modul* `AbsSet` zur Kapselung und der *Typklasse* `Ord` wie folgt aus:

```
module AbsSet(Set, incl, excl, empty, ...) where
  data Ord a => Set a = Nil | Cons !a !(Set a)
  empty = Nil
  ...
```

Eine *totale* Ordnung ist erforderlich, um ein eindeutiges *Minimum* von Mengen konstruktiv berechnen zu können. Eine partielle Ordnung oder die Typ-

klasse `Eq` würden nicht reichen. Für *endliche* Mengen muss außerdem der Konstruktor `Cons` (!) *strikt* sein.

Der Typ `set` aus der Struktur `Set` ist ein mit einer Funktion parametrisierter Typkonstruktor, der wie $<'$`SeqOrd` oder alle anderen Funktionen aus `Set` zu *instanziieren* ist, z.B. zu `set`$[$`nat`$, <]$. Eine andere Instanziierung würde einen *anderen* Typ liefern, insbesondere auch dann, wenn nur ein anderer aktueller Funktionsparameter gewählt wird: `set`$[$`nat`$, >]$. Die Instanz am Typ `set` garantiert, dass nur ebenso instanziierte Bearbeitungsfunktionen appliziert werden können.

Eine Instanz `set`$[$`nat`$, =]$ wäre ebenfalls typkorrekt; sie ist aber semantisch falsch, da die Gleichheit keine totale Ordnung ist. Die angegebene Ordnung über Mengen $<'$`Set` ist deswegen *keine Teilmengenbeziehung*, diese wäre nur partiell! Dadurch können geschachtelte Mengen `set`$[$`set`$[$`nat`$, <], <]$ gebildet werden.

Um semantische Instanziierungsfehler statisch zu vermeiden, müssten *Beweise* für geforderte Eigenschaften existieren. Die Parametrisierung könnte dann wie folgt erweitert werden:

STRUCTURE `SafeSet`$[\alpha :$ TYPE$, <: \alpha \times \alpha \to$ `bool`$,$ `p`$:$ `totalOrder`$[\alpha, <]]$

In einer konkreten Instanz müsste nun ein *benannter* Beweis für die *Eigenschaft* `totalOrder` angegeben werden (genaueres siehe Abschnitt 8.3). Ob dieser Beweis dann korrekt ist, bleibt eine andere Frage. Der unübliche dritte formale Parameter `p` *enthält* dabei den zweiten Parameter $<$, was bisher für OPAL und PVS nicht optimal realisiert wurde.

Überlagerung von Typen

Die *partiell* notierte Instanz `set`$[$`nat`$, <]$ ist *mehrdeutig*, wenn die $<$-Funktion *überlagert* ist. Mit dem ersten Argument `nat` muss aber – durch α in der formalen Parameterliste – das zweite Argument vom Typ `nat` $\times$ `nat` $\to$ `bool` sein und über diesen Typ ist dann eine Überlagerungsauflösung möglich. Falls überlagerte Funktionen *typgleich* sind, hilft eine Annotation: `set`$[$`nat`$, <'$`Nat`$]$

Auch ein Basistyp wie `nat` kann mehrdeutig sein. In einer anderen Struktur `BigNat` könnten dieselben Bezeichner gewählt worden sein:

```
STRUCTURE BigNat
  TYPE nat
  FUN <: nat × nat → bool
```

Um die gleichbezeichneten aber *verschiedenen* Typen `nat` zu unterscheiden, können sie mit ihrer Herkunft annotiert werden. Dadurch unterscheiden sich auch die Typen zu den Funktionen $<'$`Nat` und $<'$`BigNat`:

$<'$`Nat` : `nat'Nat` × `nat'Nat` → `bool`
$<'$`BigNat` : `nat'BigNat` × `nat'BigNat` → `bool`

Nach den Importen von `Nat` und `BigNat` können deswegen einige partielle Instanzen eindeutig aufgelöst werden:

```
IMPORT Nat ONLY nat <
IMPORT BigNat ONLY nat <

        Set[nat, <]              - mehrdeutig
        Set[nat'Nat, <]          - eindeutig
        Set[nat, <'Nat]          - auch eindeutig!
        Set[nat'BigNat, <'Nat]   - inkonsistent
```

Inkonsequenterweise wird die Instanz `Set`[`nat`, $<'$`Nat`] von fast allen Sprachimplementierungen als mehrdeutig zurückgewiesen, nur weil der erste Parameter `nat` mehrdeutig ist.

Dieselbe Ungenauigkeit gilt auch für die Vorlagen (*Templates*) von C++, dort werden die Parameter *sequenziell* bearbeitet; den Typen entsprechen *Klassen*:

```
template <class T, bool lt(T, T)>
class Set { ... };
```

In verschiedenen Namensräumen (`namespace A` und `B`), die in C++ selten verwendet werden, können gleichbezeichnete Klassen `C` eingeführt werden, die durch *Qualifikation* `A::C` oder `B::C` nach `using namespace A` und `B` unterscheidbar sind.

Die Spezifikationsprache LPG (*Langage pour la Programmation Générique* von Didier Bert) [BER94] unterstützt keine Überlagerung von Typen. LPG ist aber in der Lage, den aktuellen Typparameter einem aktuellen Funktionsparameter zu entnehmen. Insbesondere kann sogar auf die explizite Angabe

des Typparameters verzichtet werden, wenn dem Überlagerungen der Funktion nicht entgegen stehen. Dadurch ist z.B. die Kurznotation `Set`[$<$] möglich.

Auch in der algebraischen Spezifikationssprache CASL [BM00], die Referenzsprache der *Common Framework Initiative* (CoFI), sind Typen wie `nat` nicht überlagert; `BigNat` und `Nat` zusammen würden den Typ `nat` und die $<$-Funktion ggf. *verfeinert* oder inkonsistent spezifizieren.

5.2 Abhängige formale Parameter

Gemäß der formalen, typannotierten Parameterliste [α : TYPE, $<$: $\alpha \times \alpha \to$ `bool`] zur Struktur `Set` handelt es sich um bezüglich α *abhängige* Parameter, die in gewisser Weise mit abhängigen Typen (*dependent types*) für Funktionen vergleichbar sind.

In PVS könnte man beispielsweise die Subtraktion über den natürlichen Zahlen mit Hilfe von abhängigen Typen (`subtract`(`a` : `nat`, `b` : {`x` | `x` $\leq$ `a`}) : `nat`) *total* deklarieren. Die Abhängigkeit *auf der Ebene von Werten* ist allerdings i.A. unentscheidbar und PVS generiert bei Applikationen entsprechende Beweisverpflichtungen.

Die Abhängigkeiten zwischen Parametern können algebraisch (ebenso wie klassisch funktional) durch *Substitutionen* verwaltet werden. Dieses geschieht in Analogie zu polymorphen Funktionen, etwa :: : $\alpha \times$ `seq`[α] $\to$ `seq`[α]; die Argumente (und das Ergebnis) sind abhängig bezüglich α und Spezialisierungen sind Substitutionen für α, die in Applikationen per Unifikation berechnet werden. Das so genannte *Matchen* formaler und aktueller Parameterlisten, also inwieweit die Mehrfachvorkommen formaler Parameter konsistent durch aktuelle Parameter instanziiert werden, erfolgt analog – simultan für alle Parameter und unabhängig von ihrer Reihenfolge – durch *Unifikation*.

Die für diese Unifikation zu betrachtenden *vollständigen* Namen und *Variablen* unterscheiden sich erheblich von den einfachen funktionalen Typtermen und Typvariablen. Durch die Parametrisierung mit Funktionen treten nicht nur Typvariablen z.B. im Typ formaler Funktionsparameter auf, sondern auch *Funktionsvariablen* bzw. *Variablen mit Typ*. Stellt man die vollständigen formalen und aktuellen Parameter der Struktur `Set` gegenüber, so ist der Typ des (zweiten) aktuellen Funktionsparameters identisch zum Typ des formalen, nachdem man darauf die Substitution [α := `nat'Nat`] angewendet hat, die man dem (ersten) Typparameter entnehmen kann:

$\texttt{set}[\alpha : \text{TYPE}, <: \alpha \times \alpha \to \texttt{bool}]$ - *formal*
$\texttt{set}[\texttt{nat'Nat} : \text{TYPE}, <'\texttt{Nat} : \texttt{nat'Nat} \times \texttt{nat'Nat} \to \texttt{bool}]$ - *aktuell*

Betrachtet man im zweiten Parameter nur den Teil vor dem Typ, dann ergibt sich die Substitution $[<:= <'\texttt{Nat}]$; d.h. der (uninstanziierbare) Bezeichner $<$ für den formalen Parameter aus `Set` ist eine *Variable* (später β genannt). Diese Variable wird ersetzt durch den aktuellen Namen $<'\texttt{Nat}$ *ohne* seinen *Typ*. Ebenso ist α eine Variable, die durch den Namen `nat'Nat` *ohne* TYPE ersetzt wird. Insgesamt entsteht die aktuelle Parameterliste aus der formalen durch die Substitution $[\alpha := \texttt{nat'Nat}, <:= <'\texttt{Nat}]$.

Zur Vertiefung betrachten wir noch die folgende – ziemlich hypothetische und nicht von C++ Templates unterstützte – *formale* Parameterliste, bei der ein Funktionsparameter wiederum im Typ eines weiteren Funktionsparameters vorkommt. Die Struktur `SafeSet` aus Abschnitt 5.1 ist ähnlich; statt `totalOrder` wird jetzt der Typ `set` verwendet:

$\texttt{Foo}[\alpha : \text{TYPE},\ < : \texttt{rel}[\alpha],\ \texttt{s} : \texttt{set}[\alpha, <]]$ - *formal*

Der Typ $\texttt{rel}[\alpha]$ wird hier lediglich als abkürzendes *Synonym* (Abschnitt 8.4.3) für $\alpha \times \alpha \to \texttt{bool}$ benutzt und entsprechend ist $\texttt{rel}[\texttt{nat}]$ der Typ zur Funktion $<'\texttt{Nat}$. Eine passende Instanziierung wäre $\texttt{Foo}[\texttt{nat}, <'\texttt{Nat}, \{\}'\texttt{Set}]$, wobei die leere Menge $\{\}$ aus `Set` *generisch* importiert vorliegen soll. Die ausführlich annotierte aktuelle Parameterliste ist:

$\texttt{Foo}[\texttt{nat} : \text{TYPE},\ <'\texttt{Nat} : \texttt{rel}[\texttt{nat}],\ \{\} : \texttt{set}[\texttt{nat}, <'\texttt{Nat}]]$ - *aktuell*

Die Substitution $[\alpha := \texttt{nat'Nat}, <:= <'\texttt{Nat}]$ durch die (unveränderten) ersten beiden Parameter liefert nun den aktuellen Typ für den dritten Parameter, und aus der Typinstanz $\texttt{set}[\texttt{nat}, <'\texttt{Nat}]$ ergibt sich direkt – durch die Deklaration FUN $\{\} : \texttt{set}$ in `Set` – die Instanz $\{\}[\texttt{nat}, <'\texttt{Nat}]$ (vgl. Abschnitt 5.4.3). Die Substitution für den dritten Parameter lautet daher $[\texttt{s} := \{\}[\texttt{nat}, <'\texttt{Nat}]]$. Dieses letzte Beispiel soll den Begriff *Namensinstanz* (vgl. `inst` in Abschnitt 5.3) für vollständige *Namen ohne Typ* motivieren, obwohl er für unparametrisierte Namen (aus `Nat`) nicht sehr treffend ist.

Vollständige formale Parameter sind also keine elementaren Typvariablen wie in klassisch funktionalen Sprachen sondern *Variablen mit Typ* und die Variablen stehen nicht für vollständige Namen sondern nur für die Namen *ohne Typ*. Eine aktuelle Parameterliste *passt* also auf eine formale Parameterliste, wenn die *Typen* korrespondierender Parameter *identisch* sind oder durch die Substitution von *untergeordneten* Parametern identisch werden.

Die *Variablen* der formalen Parameterliste, für die zwecks Instanziierung eine Substitution gesucht wird, sind genau die benutzerdefinierten Bezeichner *ohne Typ*, d.h. $\alpha, <$ und `s` aus dem Kopf der Struktur `Foo`$[\alpha, <, \mathtt{s}]$ ohne Typannotationen. Der Typ eines (übergeordneten) formalen Parameters kann – da Variablen atomar sind – einen untergeordneten Parameter nur ganz oder gar nicht *enthalten* und alle Parameter einer Struktur sind bezüglich dieser Relation partiell geordnet.

5.3 Vollständige Namen

Die *vollständigen Namen* für die algebraischen Sprachen sind anders als die einfachen Typterme aus Abschnitt 2.1.2 wie folgt aufgebaut:

```
name ::== inst : type            - Namensinstanz und Typ

inst ::== ide[name, ..., name]   - instanziierter Bezeichner
          var                    - Variable

type ::== TYPE
          inst : TYPE            - Funktionstyp
```

Namen bestehen also aus zwei Teilen, ihrer Namensinstanz und ihrem Typ, und sind vor allem *keine* Variablen. Variablen stehen nur für *Namensinstanzen* `inst`, das sind einfache (mit Herkunft annotierte) Bezeichner `ide` mit einer je nach Parametrisierung passenden *Instanzliste* von (vollständigen) Namen.

Die Namensterme sind *frei generiert* und *initial*. (Die *Grammatiken* hier entsprechen den Datentypen aus *strikten* Sprachen wie OPAL, PVS und ML, aber nicht HASKELL.) Die ersten Namensinstanzen `inst` sind daher *unparametrisiert* mit leeren Instanzlisten und erst damit kann man *echt* instanziierte Namen bilden. Darüberhinaus gibt es keine zyklischen Namen, beispielsweise kann sich der *partiell notierte* Name $<[\mathtt{nat}, <]$ nicht selbst enthalten, weil der zugehörige vollständige Name unendlich wäre: $<[\mathtt{nat}, <[\mathtt{nat}, <[\ldots]]]$. Der hierarchische und zyklusfreie Aufbau der Namen wird in Kapitel 8 ausgenutzt, um Deklarationen schrittweise und inkrementell, aber unabhängig von ihrer textuellen Position zu analysieren.

Namen sind entweder Typen oder Funktionen (mit ihrem Typ). Insbesondere die zweistelligen Typkonstruktoren $\rightarrow$ und $\times$ kann man sich als generische Typbezeichner aus einer mit zwei Typen parametrisierten Struktur

vorstellen, wobei von einer systematischen Präfix-Notation (wie $\times[\alpha, \beta]$) aus Lesbarkeitsgründen abgewichen wird.

Genau genommen ist für Funktionen die kürzere Repräsentation `inst : inst` an Stelle von `inst : inst :` TYPE ausreichend. Wenn man das Schlüsselwort TYPE als spezielle Namensinstanz betrachtet, dann wäre sogar insgesamt eine Namensmodellierung `name ::== inst : inst` möglich, die allerdings einen unsinnigen Namen, der mit TYPE beginnt, nicht ausschließen würde.

Für eine noch kürzere Notationskonvention kann auf die explizite Typannotation mit TYPE ganz verzichtet werden; Typnamen wären dann an fehlenden Typannotationen erkennbar. Die Modellierung als Grammatik sähe dann wie folgt aus:

```
name ::== inst          - Typname
          inst : inst   - Funktionsname
inst ::== ...           - wie oben
```

Wenn man allerdings sowohl *vollständige* als auch *partielle* Namen (aus Abschnitt 5.4) zusammen betrachtet, dann kann man einen vollständigen Typnamen nicht von einem partiellen Namen ohne Typannotation unterscheiden.

5.3.1 Substitution

Substitutionen sind endliche Abbildungen von nummerierten Variablen auf die *Namensinstanzen* `inst`. (Statt `var(n)` wird $\alpha, \beta, \ldots$ geschrieben.) Die Unifikation zweier vollständiger Namen, einer mit formalen und der andere mit aktuellen Parametern, liefert bei einer korrekten Instanziierung eine Substitution, die angewendet auf den formalen Namen exakt den Ersetzungsprozess von vollständigen formalen Parametern durch *passende* vollständige aktuelle Parameter widerspiegelt:

```
set[α : TYPE, β : rel[α]]          - formal
set[nat : TYPE, <'Nat : rel[nat]]  - aktuell
```

Statt aber ganze Teilbäume, etwa β : `rel`$[\alpha]$ durch $<'$`Nat : rel[nat]`, zu ersetzen, werden nur Variablen, also einfache Blätter substituiert. Beim sequenziellen Ersetzen ganzer Teilbäume müsste die Reihenfolge beachtet werden: am einfachsten wäre es, erst β : `rel`$[\alpha]$ durch $<'$`Nat : rel[nat]` und anschließend α : TYPE durch `nat` : TYPE zu ersetzen; würde dagegen zuerst α : TYPE durch `nat` : TYPE ersetzt, dann müsste auch der Typ des zweiten formalen

Parameters von `rel`$[\alpha:$ TYPE$]$ zu `rel`$[$`nat` : TYPE$]$ geändert und anschließend der *veränderte* Teilbaum $\beta:$ `rel`$[$`nat`$]$ durch $<'$`Nat` : `rel`$[$`nat`$]$ ersetzt werden.

Der parallele Ersetzungsprozess von ganzen Teilbäumen ist also viel komplizierter als die bekannte (parallele oder sequenzielle) Substitution von atomaren Variablen. Gewöhnungsbedürftig ist lediglich, dass man einer Variablen nicht direkt ansieht, ob sie für einen Typ oder eine Funktion steht. Diese Rolle ergibt sich allein aus ihrer Position im vollständigen Namensterm. Variablen stehen dabei immer typkonform für Namensinstanzen.

5.3.2 Unifikation

Die Substitutionen für die Variablen ergeben sich durch *Unifikation* der *vollständigen* formalen und aktuellen Namen *mit Typ*. Für den Typparameter α wird damit z.B. gewährleistet, dass der aktuelle Name `nat` tatsächlich ein Typ (und keine Funktion) ist. Die Unifikation des vollständigen Namens $<'$`Nat` : `rel`$[$`nat`$]$ mit dem vollständigen formalen Parameter $<:$ `rel`$[\alpha]$ liefert nicht nur eine Substitution für die Variable $<$, sondern – über den Typ – auch für die Variable α und garantiert so die Konsistenz (und später eine Überlagerungsauflösung) für den Typparameter. Bei einer Abarbeitung der Parameter *in beliebiger Reihenfolge* werden die Substitutionen entsprechend komponiert.

Beim Namen $\{\}$ für die leere Menge aus der parametrisierten Struktur `Set` werden die Variablen wie bei polymorphen Funktionen *gebunden*. Eine fast vollständige Notation, bei der die Benennung der gebunden Variablen irrelevant ist (β statt $<$), wäre folgende:

$$\forall\alpha\ \beta.\ \{\}'\texttt{Set}[\alpha:\ \text{TYPE}, \beta:\ \texttt{rel}[\alpha]]\ :\ \texttt{set}'\texttt{Set}[\alpha:\ \text{TYPE}, \beta:\ \texttt{rel}[\alpha]]$$

Zum Zwecke der Auflösung der Instanz `Foo`$[$`nat`$, <, \{\}]$ wird der generische Name $\{\}$ mit *frischen Unbekannten* instanziiert: $\{\}[\gamma, \delta]$: `set`$[\gamma, \delta]$. Dieser Namenskandidat wird nun mit dem formalen Parameter `s` : `set`$[\alpha, <]$ der Struktur `Foo` *unifiziert*. Die Unifikation beider `set`-Typen liefert eine Substitution $[\alpha := \gamma, <:= \delta]$, was zusammen mit $[\alpha :=$ `nat`$, <:= <'$`Nat`$]$ durch die ersten beiden Parameter die konkrete Typinstanz `set`$[$`nat`$, <'$`Nat`$]$ liefert und damit die Namensinstanz $\{\}[$`nat`$, <'$`Nat`$]$. Für den *generisch* importierten Namen $\{\}$ aus der Struktur `Set` ergibt sich also per Unifikation die konkrete Instanz.

Gemäß dem *Annotationskonzept* von OPAL besteht ein Name aus *vier* Tei-

len: seinem Bezeichner, seiner Herkunft, seiner Instanz und seinem Typ. Ein Name ist vollständig bekannt, wenn alle Teile vollständig bekannt sind. Bei einem *partiellen* Namen sind die Herkunft, die Instanz und der Typ *optionale* Annotationskomponenten; nur der einfache Bezeichner (ganz links vorne) ist obligatorisch. Diese Zerlegung in vier Teile führte in [Reu98] dazu, dass statt einer *drei* Variablen betrachtet wurden: getrennt (und typverschieden) für Bezeichner, Herkunft und Instanz. Dass keine Variable für den Typ nötig ist, wurde dabei korrekt erkannt. Weniger umständlich ist die hier angegebene Zerlegung vollständiger Namen in *zwei* Teile: in *Namensinstanz und Typ*.

5.4 Partielle Namen

Die vom Algorithmus zu identifizierenden *partiellen* Namen unterscheiden sich von den vollständigen darin, dass Typannotationen sowie Instanz- und Herkunftsannotationen *optional* sind. Die *Herkunft* zur Reduzierung von Überlagerungen sei hier durch die Bezeichner `ide` behandelt:

`name` ::==	`inst`	- *Namensinstanz*
	`inst` : `type`	- *typannotierter Name*
`inst` ::==	`ide`[`name`, . . . , `name`]	- *instanziierter Bezeichner*
	`ide`	- *einfacher Bezeichner*
`type` ::==	TYPE	- *Typ eines Typs*
	`inst`	- *Funktionstyp*
	`inst` : TYPE	- *Funktionstyp mit* TYPE

Ein Typname darf (höchstens) mit TYPE und ein Funktionsname nur mit einem Typnamen typannotiert werden. Eine Modellierung der Funktionstypen durch `type` ::== `name` würde zu tief geschachtelte Typannotationen der Form `inst` : `inst` : `inst` ermöglichen. Die beiden Notationsvarianten `inst` : `inst` und `inst` : `inst` : TYPE sind (wieder) äquivalent.

Für partielle Namen ist der Unterschied zwischen Bezeichnern mit und ohne Instanz wichtig. Vollständige Namen müssen immer korrekt instanziiert vorliegen, d.h. *unparametrisierte* Namen werden – trivial instanziiert – mit einer *leeren Instanzliste* repräsentiert. Ein bloßer Bezeichner `ide` als Notationsalternative für den *vollständigen* unparametrisierten Namen `ide`[ ] ist im Zusammenhang mit partiellen Namen problematisch (und nur im Kontext von ausschließlich vollständigen Namen akzeptabel); für *partielle* Namen bedeu-

tet eine Bezeichnung `ide` *ohne* Instanzliste, dass die Instanziierung *unbekannt* ist und durch die Namensidentifikation ermittelt werden muss, während die Notation *mit* Instanzliste für eine bezüglich der Parameteranzahl passende Parametrisierung steht. Der (in OPAL ausgeschlossene) partielle Name `ide[]` könnte insofern als ein garantiert *unparametrisierter* Name betrachtet werden.

Ein Bezeichner wie z.B. `set` steht potenziell für alle vollständigen aber *monomorphen* Namen mit demselben Bezeichner, die einem Namensraum entnommen werden können. (Namensräume kann man sich als Mengen von vollständigen Namen vorstellen, wobei generische Namen mit Hilfe von gebundenen Variablen repräsentiert werden.) Für unbekannte Namensinstanzen werden *frische* und *freie* Variablen für spätere Substitutionen eingeführt.

Eine explizite Notation von Variablen ist mit partiellen Namen nicht möglich. Statt Instanzannotationen wegzulassen, kann durch die *zusätzliche Notationsvariante* `inst` := _, die in Abschnitt 6.1.2 näher erläutert wird, ein *Unterstrich* innerhalb von Instanzen benutzt werden. Auch damit werden Variablen nur implizit notiert, jeder Unterstrich steht für eine *neue* Unbekannte.

5.4.1 Redundante Typannotationen

Typannotationen für partielle Namen sind streng genommen überflüssig, da der Typ eines Namens immer *implizit* bekannt ist. Bei einer Funktionsdeklaration FUN `f` : `set`[`nat`, <] ist klar, dass der *Typname* `set`[`nat`, <] : TYPE zu identifizieren ist. Ebenso ist der Typ für die Ordnungsrelation < in der Instanz `set`[`nat`, <] implizit durch die Kenntnis der formalen Parameter von `set` als `rel`[α : TYPE] – oder schon spezialisiert als `rel`[`nat'Nat` : TYPE] – gegeben.

Eine *explizite* Typannotation, etwa < : `rel`[`nat`] für das Instanzargument, erfordert sogar eine Zusatzanalyse (Abschnitt 6.1.3), da die explizite Typannotation natürlich zum *impliziten* Typ kompatibel sein muss. Dafür erlaubt die Redundanz der Typannotationen eine flexiblere Notation: die TYPE-Annotation in `seq`[`nat` : TYPE] kann bei überlagerten Namen ∀`n`.`seq`[`n` : `nat`] und `nat` : `nat` eine Mehrdeutigkeit mit `seq`[`nat` : `nat`] vermeiden. Insgesamt kann sogar die *Überlagerung für Strukturen* `Seq` aufgelöst werden.

Für global eindeutige Strukturen ist ein Sprachentwurf *ohne Typannotationen* plausibel, da sich implizite Typen immer *vollständig instanziiert* aus

dem *formalen* Parametertyp (z.B. `rel`$[\alpha]$) und aktuellen, abhängigen aber *untergeordneten* Parametern (wie `nat` für α) ergeben. Eine *Herkunftsannotation* am instanziierten Namen `set'Set`$[$`nat`$, <]$ garantiert bei Bedarf eine *eindeutige* formale Parameterliste!

Konkret in PVS sind Typannotationen für *Namen* nicht vorgesehen; der Unterschied zu *Namensfronten* entfällt. Dennoch können typverschiedene Namen als *Ausdrücke* typannotiert werden und in PVS darf man mit Ausdrücken *instanziieren* (vgl. Abschnitt 5.5).

5.4.2 Überlagerung innerhalb einer Struktur

Die Identifikation instanziierter Namen basiert einzig und allein auf den unterschiedlichen Typen für überlagerte Namen. Für *typgleiche* Namen sind die Typannotationen gleich, daher müssen überlagerte typgleiche Namen mit einer *Herkunft* oder *Instanz* annotiert werden und dadurch unterscheidbar sein. Überlagerte Namen aus *derselben* Struktur müssen typverschieden sein, da die Herkunft und mögliche Instanzen keine Unterscheidung mehr erlauben. Genau diese minimale Beschränkung der Überlagerung ist – typisch algebraisch – für PVS und OPAL verwirklicht. (In OPAL werden typgleiche Deklarationen als *Wiederholungen* ignoriert.)

Da *Typnamen* den eindeutigen *Typ* TYPE haben, können sie innerhalb einer Struktur nicht überlagert werden. Für Typen von überlagerten *Funktionen* aus *parametrisierten* Strukturen gibt es eine weitere Beschränkung, die zwar als Kontextbedingung für OPAL [Gro94] bekannt ist, aber nicht durch die Kontextanalyse behandelt wird. Die Typen überlagerter Funktionen einer Struktur müssen *für alle Instanzen* verschieden sein. In der folgenden Struktur `Pair` sind zunächst die Typen für die fälschlich überlagerten Selektoren `get` verschieden:

```
STRUCTURE Pair[α, β]
  TYPE α
  TYPE β                  - Typparameter
  TYPE pair = ...         - Datentyp
  FUN get : pair → α
  FUN get : pair → β      - Selektoren
```

Außerhalb von `Pair` sind die formalen Parameter nicht bekannt und können nicht in Typannotationen verwendet werden. Nach folgendem Import fallen daher die *instanziierten* Typen der `get`-Funktion zusammen:

IMPORT `Pair[nat, nat]` ONLY `pair get`

... `get : pair[nat, nat] → nat` - *1. oder 2. Komponente?*

Üblicherweise wird diese Problematik implementierungstechnisch ignoriert. Dadurch kann eine unauflösbare Mehrdeutigkeit entstehen, die der Benutzer vermeiden muss, indem er für den obigen Fall *verschiedene* Bezeichner (z.B. `1st` und `2nd` innerhalb von `Pair`) wählt. Besser ist es aber, durch die *Kontextanalyse* die Überlagerung innerhalb von Strukturen auf solche Funktionen zu beschränken, deren Typen nicht nur verschieden, sondern bezüglich der formalen Parameter *nicht unifizierbar* sind. Damit ist gewährleistet, dass *für jede Instanziierung* die Typen verschieden bleiben. (Umgekehrt beschreibt eine erfolgreiche Unifikation, wie die Typen zusammenfallen können.)

Eine denkbare Analyse zusammenfallender Funktionstypen erst beim konkreten Import `Pair[nat, nat]` wäre fast ebenso aufwendig, käme aber zu spät, weil nun eine Korrektur in der Struktur `Pair` alle anderen Strukturen betreffen könnte, die `Pair` importieren. Den Import von `Pair[nat, nat]` zu verbieten, wäre natürlich ebenso unbefriedigend, wie sich mit der Unbenutzbarkeit von zusammenfallenden Namen `get` abzufinden.

Algebraisch können prinzipiell verschiedene typkonforme formale Parameter mit denselben aktuellen Parameter instanziiert werden. Der eher zufällige Effekt einer Beschränkung für Instanziierungen durch zusammenfallende Funktionsnamen ist insofern eher als Fehler denn als (unalgebraisches) *Feature* zu bewerten.

Auch die Entwurfsoption, Annotationen mit dem *uninstanziierten* Originaltyp `get : pair` $\rightarrow \alpha$ *außerhalb* von `Pair` zu erlauben, hat Nachteile und ist in PVS und OPAL illegal:

- Die Abstraktion der Strukturparameter ist nicht vollständig, weil formale Parameter für spezielle Zwecke global sichtbar bleiben.
- Die Typannotationen für Namen und Ausdrücke unterscheiden sich. Ein (partiell notierter) Typ zum Ausdruck muss *instanziiert* sein!

5.4.3 Inferierbarkeit der Instanz

Für Typen ist offensichtlich, dass sie in den meisten Fällen nicht ohne Instanzen auskommen. TYPE-Annotationen sind entbehrlich. Für Funktionen

kann man sich fragen, wann sich aus der Kenntnis von Herkunft und Typ eine *Instanz* berechnen lässt, wie das beispielsweise für {} in `Foo`[`nat`, $<$, {}] geschah. Ein typisches Gegenbeispiel wäre aber eine Funktion, deren Typ nicht von allen formalen Parametern abhängt:

STRUCTURE `Seq`[α]
...
FUN `foo` : `nat` $\rightarrow$ `nat`

Im Typ der Funktion `foo` kommt der Typparameter α nicht vor, daher kann man dem Typ auch nicht den aktuellen Parameter entnehmen und sowohl OPAL als auch PVS erwarten deswegen eine explizite Instanzannotation.

Die Funktion `foo` ist ein vermeidbarer Spezialfall, der die Frage aufwirft, ob die Spezifikation oder die Definition dieser Funktion vom formalen Parameter α abhängt. Solange eine Struktur nur mit Typen parametrisiert ist, kann sich das operationale Verhalten für mehrere Instanziierungen durch eine bezüglich der formalen Typparameter *uniforme* Definitionsgleichung nicht unterscheiden. Insofern gehört die Funktion `foo` eigentlich in eine *unparametrisierte* Struktur und könnte als Fehler diagnostiziert werden oder einfach als beliebig instanziiert betrachtet werden.

Bei der Parametrisierung mit Funktionen dagegen kann es durchaus sinnvoll sein, wenn der Typ einer Funktion unabhängig vom Funktionsparameter ist. Dieses ist z.B. für die Listenordnung $<'$`SeqOrd` der Fall. Der Typ dieser Funktion hängt nur vom Typparameter α ab, für die Instanziierung wird aber zusätzlich eine Elementordnung β benötigt:

$\forall\alpha\ \beta.\ <'$`SeqOrd`$[\alpha, \beta$: `rel`$[\alpha]] :$ `seq`$[\alpha] \times$ `seq`$[\alpha] \rightarrow$ `bool`

Da ein Funktionsparameter ein Zusatzargument ist, könnte man den Typ der *generischen* Funktion $<'$`SeqOrd` wie für eine Funktion höherer Ordnung repräsentieren:

$\forall\alpha\ \beta.\ <'$`SeqOrd`$[\alpha, \beta$: `rel`$[\alpha]] :$ `rel`$[\alpha] \rightarrow$ `seq`$[\alpha] \times$ `seq`$[\alpha] \rightarrow$ `bool`

Zwar kommt dadurch der Funktionsparameter β nicht im Typ vor, er kann sich aber außer durch eine explizite Instanziierung, die den Typ wieder adaptieren müsste, auch aus einer konkreten Applikation in einem Ausdruck wie $<'$`SeqOrd`$(<'$`Nat`) ergeben. Dieser *neue* Aspekt wird nochmals in Abschnitt 5.5 erläutert.

Eine Abhängigkeit von *allen* formalen Parametern ist aber garantiert immer dann gegeben, wenn die Funktionalität einer Funktion einen Typ aus *derselben* Struktur umfasst. Dazu erweitern wir die Struktur `Seq` noch um folgende überlagerten Funktionen:

```
STRUCTURE Seq[α]
  TYPE α                  - Typparameter
  TYPE seq = ...          - Datentyp

  FUN seq: α → seq
  FUN seq: α × α → seq
```

Der Bezeichner `seq` ist dreifach überlagert; es gibt zwei Funktionen und einen Typ. Der annotierte Namen `seq'Seq[nat]` ist damit (nach Import) *mehrdeutig*, wird aber eindeutig, wenn der Typ gegeben ist! Der typannotierte Name `seq: nat × nat → seq` ist nur deswegen mehrdeutig, weil am *Ergebnistyp* die Instanz `seq[nat]` bzw. `seq[nat × nat]` fehlt, die der Funktionsinstanz entspricht.

Ohne einen Typ aus derselben Struktur und obwohl alle formalen Parameter in den Funktionalitäten überlagerter Funktionen vorkommen, kann die Instanz nicht immer dem Typ entnommen werden:

```
STRUCTURE Id[α]
  FUN id: α → α
  FUN id: α × α → α × α
```

Der typannotierte Name `id: nat×nat → nat×nat` passt zur Instanz `id[nat]` und `id[nat×nat]`. Gleichzeitig gibt es keine Instanz, für die beide Funktionen zusammenfallen würden; der Typ der zweiten Funktion ist immer länger als der der ersten. Um solche Fälle auszuschließen, muss man die Unifikation der Typen von überlagerten Funktionen mit *verschiedenen Variablensätzen* durchführen. Die Typen $\alpha \to \alpha$ und $\beta \times \beta \to \beta \times \beta$ sind für `id'Id` durch die Substitution $[\alpha := \beta \times \beta]$ unifizierbar, während obige Typen $\alpha \to$ `seq`$[\alpha]$ und $\beta \times \beta \to$ `seq`$[\beta]$ nach wie vor – durch den *Occurs-Check* $\beta \prec \beta \times \beta$ am Typ `seq` – *nicht* unifizierbar sind.

Zusammenfassend wird hier also vorgeschlagen, die Überlagerungsauflösung für Funktionen einer Struktur allein mit Hilfe der vollständig instanziierten Typen zu garantieren. Die Beschränkung für Überlagerungen innerhalb von Strukturen wird dadurch nur unwesentlich verschärft und implementierungstechnisch besteht ebenfalls kaum ein Unterschied. Unbenutzte Typparameter (vgl. `foo'Seq`) können auf Grund der Uniformität ignoriert werden. Die

Instanziierung von Funktionsparametern sollte als eine *Applikation* wie bei Funktionen höherer Ordnung betrachtet werden. Auf eine *optionale* Annotation von Instanzen möchte man natürlich nicht verzichten, da diese in vielen Fällen kürzer und angemessener als Typannotationen sind. Wenn aber ein Typ eindeutig durch den Kontext gegeben ist, dann möchte man möglichst nicht noch zu einer Instanzannotation *verpflichtet* werden.

5.5 Funktionen höherer Ordnung

Die Parametrisierung mit Funktionen wird nun noch mit den üblichen Funktionen höherer Ordnung verglichen. Die Sequenzordnung $<'$`SeqOrd` könnte auch, statt mit einem Funktionsparameter zur Struktur, direkt mit einem expliziten Funktionsargument deklariert werden:

STRUCTURE `SeqOrd2`$[\alpha]$
$\quad\vdots$
FUN $<$: $(\alpha \times \alpha \to$ `bool`$) \to$ `seq`$[\alpha] \times$ `seq`$[\alpha] \to$ `bool`

Die Deklaration von $<'$`SeqOrd2` hat den Nachteil, dass sie für Instanziierungen nicht benutzt werden kann. Aktuelle Funktionsparameter dürfen (in OPAL) nur instanziierte Funktionsnamen sein, aber keine Ausdrücke wie die Applikation $<(<)$:

`set`[`seq`[`nat`], $<'$`SeqOrd2`($<'$`Nat`)] - *illegal*
`set`[`seq`[`nat`], $<'$`SeqOrd`[`nat`, $<'$`Nat`]] - *legal*

In diesem Zusammenhang ist die Notationsalternative für Instanzen aus LPG [BER94] erwähnenswert, bei der aktuelle Funktionsparameter ausreichen, wenn Funktionsparameter Typparameter enthalten. Eine Instanz für Mengen (siehe auch Abschnitt 8.3) könnte so als `set`$[<]$ notiert werden und ein *Name* $<[<]$ wäre syntaktisch sehr ähnlich zur *Applikation* $<(<)$. Ganz pragmatisch wäre es, Applikationen als Instanziierungen zu interpretieren:

`set`[`seq`[`nat`], $<'$`SeqOrd`($<'$`Nat`)] - *neu*

Die generisch importierte Funktion $<'$`SeqOrd` wird dabei mit dem *höheren* Typ `rel`$[\alpha] \to$ `rel`[`seq`$[\alpha]$] betrachtet. In PVS ist die Instanziierung von Funktionsparametern mit beliebigen Ausdrücken möglich. Die Typgleichheit für Instanzen `t`[$\mathtt{e}_1$] und `t`[$\mathtt{e}_2$] hängt dort von der Gleichung $\mathtt{e}_1 = \mathtt{e}_2$ ab und ist deswegen i.A. unentscheidbar. Bei der Namensgleichheit von OPAL und LPG

sind die Typen $\mathtt{t}[\mathtt{n}_1]$ und $\mathtt{t}[\mathtt{n}_2]$ verschieden, obwohl nicht ausgeschlossen ist, dass die Funktionsnamen oder Konstanten $\mathtt{n}_1$ und $\mathtt{n}_2$ *semantisch gleich* und nur unterschiedlich bezeichnet sind. Insofern könnte man auch die Instanzen $\mathtt{t}[\mathtt{e}_1]$ und $\mathtt{t}[\mathtt{e}_2]$ einfach als verschieden werten, wenn die Ausdrücke $\mathtt{e}_1$ und $\mathtt{e}_2$ *syntaktisch* verschieden sind. (Die syntaktische Gleichheit ist unabhängig von der Benennung gebundener λ-Variablen.) Prinzipiell sollte auch die Instanziierung mit einer *lokalen* λ-Variablen möglich sein, die dabei ihren Gültigkeitsbereich nicht verlassen kann: $(\lambda < .\{\}[\mathtt{nat}, <])$ wäre illegal, weil der Typ $\mathtt{rel}[\mathtt{nat}] \to \mathtt{set}[\mathtt{nat}, <]$ die λ-Variable $<$ enthält.

Wozu benötigt man also noch Funktionen höherer Ordnung? In CASL, LPG und OBJ wird darauf verzichtet. Auf die Instanziierung mit einem Funktionstyp $\mathtt{seq}[\mathtt{nat} \to \mathtt{nat}]$ oder eine Funktion als Datentypkomponente TYPE $\mathtt{funct} = \mathtt{abs}(\mathtt{rep} : \alpha \to \beta)$ möchte man i.d.R. aber nicht verzichten. Der Funktionstyp sollte gleichberechtigt zu anderen Typkonstruktoren ein *first-class Citizen* sein. Darüberhinaus kompensiert die algebraische Parametrisierung nur Funktionen *zweiter* Ordnung, da die Parameter von maximal erster Ordnung sind. Für Funktionen *erster* Ordnung würden *Konstanten* mit Parametrisierung ausreichen:

```
STRUCTURE Nat2[a : nat, b : nat]
  FUN < : bool                    - unüblich
```

Die *curried* Schreibweise für Parameter `Nat2[a : nat][b : nat]`, die in CASL möglich ist, ändert nichts an der ersten Ordnung, erlaubt aber eine *Teilinstanziierung*. Trotzdem würde man Funktionen nie so deklarieren wollen, da alle Funktionen einer Struktur dieselbe *Stelligkeit* haben müssten. Durch *überlagerte* Strukturen könnte dieser Nachteil aber kompensiert werden.

Neu wäre es, nur für die formalen Strukturparameter Funktionen höherer Ordnung zuzulassen, aber sich innerhalb der Struktur auf Funktionen erster (oder sogar nullter!) Ordnung zu beschränken. Als aktuelle Parameter müssen dann *uninstanziierte* Funktionen verwendbar sein: $\mathtt{set}[\mathtt{nat}, <'\mathtt{Nat2}]$.

Ebenso wie ein Typkonstruktor innerhalb einer Struktur als unparametrisiert betrachtet wird, ist eine Funktion wie $<$ innerhalb von `Nat2` eine Konstante; nach einem generischen Import stehen aber potenziell polymorphe Typen und Funktionen höherer Ordnung zur Verfügung und können wie in klassisch funktionalen Sprachen *appliziert* werden. Ein konsequenter Sprachentwurf in diese Richtung, der λ-, LET- und FIX-Ausdrücke berücksichtigt oder verwirft, ist noch offen. Die konsequente Beschränkung auf exportierte Konstanten ist vermutlich nicht möglich, da durch gekapselte Funktionstypen (wie `funct`)

der Begriff Konstante und Funktion verschwimmt. (Tatsächlich werden die Funktionen in PVS *Konstanten* genannt.)

Namensinstanziierung und Funktionsapplikation können aber zusammenfallen und Funktionen höherer Ordnung werden ein Teilaspekt der Generizität durch Parametrisierung. Die parametrisierte Form <'`SeqOrd` kann universeller als die klassische alternative Form <'`SeqOrd2` appliziert werden. Damit umfasst die Namensidentifikation die Typanalyse von Applikationen!

Mit Subtyp-Polymorphie wird allerdings die so genannte *Typresolution* für polymorphe Funktionen höherer Ordnung *unentscheidbar* [Naz95]. Diesem Problem wird in CASL und OBJ, die Subtypen unterstützen, aus dem Weg gegangen. (OBJ und SML unterstützen nicht die elegante Instanziierung mit korrespondierenden Parameterlisten.) Für CASL ist die Ergänzung um höhere Ordnung ein Forschungsthema. In PVS werden Unentscheidbarkeitsprobleme zu Beweisverpflichtungen:

Algebraische Sprache	Funktionen höherer Ordnung	Subtyp-Polymorphie
PVS	ja	ja
SML	ja	nein
OPAL	ja	nein
CASL	nein	ja
OBJ	nein	ja
LPG	nein	nein

Kapitel 6

Namensidentifikation

Die Aufgabe der Namensidentifikation ist es, ausgehend von einem Namensraum (Kapitel 7) bestehend aus generischen und überlagerten vollständigen Namen, partielle Namen aus dem Quelltext zu identifizieren, d.h. ihnen einen eindeutigen, vollständigen und *instanziierten* Namen zuzuordnen.

Diese Namensidentifikation kann in Analogie zur algebraischen Typanalyse gemäß Algorithmus $\mathcal{W}_o$ (Abschnitt 4.1) erfolgen. Statt am Typ ist man an der Instanz interessiert. Für die Namensidentifikation spielt nur das Nachsehen im Namensraum, der (VAR)-Fall, und die Applikation (APPL) eine Rolle. Der Namensraum selbst bleibt bei der ganzen Analyse unverändert und enthält keine freien Variablen. (Die Variablen in generischen Namen sind gebunden.)

Eine Instanz $\texttt{set}[\texttt{nat}, <]$ mit einer korrekten Anzahl von Argumenten ist dabei – *curried* notiert $\texttt{set}[\texttt{nat}][<]$ – wie eine linksassoziativ geklammerte Applikation zu analysieren. Die durch die Teilapplikationen gewonnenen Substitutionen werden direkt für die Auflösung weiterer Argumente berücksichtigt (und die Wahrscheinlichkeit einer kombinatorischen Explosion wird etwas reduziert). Die Reihenfolge der Argumente ist beliebig, solange nur formale und aktuelle Parameter korrespondieren.

Bei einer bottom-up Analyse würden für einen instanziierten Bezeichner $\texttt{i}[\texttt{p}_1, \ldots, \texttt{p}_\texttt{k}]$ zunächst unabhängig die Kandidaten für den Bezeichner $\texttt{i}$ und für die Namen $\texttt{p}_\texttt{j}$ mit $\texttt{1} \leq \texttt{j} \leq \texttt{k}$ bestimmt, wobei die Kandidaten für $\texttt{i}$ direkt dem Namensraum entnommen und die Kandidaten für die $\texttt{p}_\texttt{j}$ rekursiv berechnet würden. Erst danach wären alle Kombinationen der Kandidaten (mit disjunkten freien Variablen) auf Kompatibilität bezüglich der Parametrisierung (Abschnitt 5.2) zu prüfen.

Der folgende Algorithmus $\mathcal{I}$ zur Namensidentifikation orientiert sich am top-down Algorithmus $\mathcal{M}$ (Abschnitt 2.5). Dabei fließen die Beschränkungen durch die Wahl eines Kandidaten z.B. für den Toplevel-Bezeichner `i` in die *abhängige* Berechnung der Kandidaten für die Instanzliste $[\mathtt{p}_1, \ldots, \mathtt{p}_k]$ ein. Bei mehreren Kandidaten für einen überlagerten Bezeichner `i` werden dabei allerdings wie bei der top-down Cormack-Überlagerungsauflösung (Abschnitt 3.1.2) ggf. einige Kandidaten für die Instanzliste mehrfach berechnet.

Das Ergebnis von Algorithmus $\mathcal{I}$ ist eine Liste von Substitutionen (Abschnitt 6.2), die – angewendet auf den a-priori Namen – *alle* vollständigen und passenden Namenskandidaten beschreiben. Ein Vorteil der Namensidentifikation ist, dass ein Namensraum *nicht abgeschlossen* sein muss, d.h. *offen* sein darf (Abschnitt 6.3). Außerdem wird eine gewisse Monotonie erfüllt (Abschnitt 6.4), die später für eine *inkrementelle* Namensraumkonstruktion wichtig ist (Abschnitt 8.1).

6.1 Identifikationsalgorithmus $\mathcal{I}$

Die Namensidentifikation einzelner Namen und ganzer Instanzlisten erfolgt wechselseitig rekursiv. Die Verwaltung der Zähler $(\mathtt{l}, \mathtt{m}, \mathtt{n})$ für frische Variablen erfolgt analog zum Algorithmus $\mathcal{W}_o$. (Für einen einzigen *globalen* Zähler, müsste man mit dem Maximum jeder Liste weiter rechnen.) Zur Unterscheidung von *partiellen* und *vollständigen* Namen seien die zugehörigen Datentypen in unterschiedlichen *Herkunftsstrukturen* deklariert: `Syn` für die partielle konkrete Syntax und `Ast` für die vollständige abstrakte Syntax. Der vollständige *a-priori Namen* $\mathtt{f} : \mathtt{name'Ast}$ ist am Anfang eine *freie* Variable mit dem Typ TYPE. Zunächst wird nur der prinzipiell ausreichende Fall (Abschnitt 5.4.1) für partielle Namen *ohne Typ* `inst'Syn` vorgestellt:

FUN $\mathtt{I} : \mathtt{env} \times \mathtt{inst'Syn} \times \mathtt{name'Ast} \times \mathtt{nat} \to \mathtt{seq}[\mathtt{subst} \times \mathtt{nat}]$
FUN $\mathtt{I} : \mathtt{env} \times \mathtt{seq}[\mathtt{name'Syn}] \times \mathtt{seq}[\mathtt{name'Ast}] \times \mathtt{nat} \to \mathtt{seq}[\mathtt{subst} \times \mathtt{nat}]$

$$
\begin{aligned}
&\mathtt{I}(\mathtt{C},\mathtt{i},\mathtt{f},\mathtt{n}) = \{(\mathtt{S}, \mathtt{n} + \#(\mathtt{args}(\mathtt{c}))) \mid \\
&\quad \mathtt{c} \in \mathtt{C} \wedge \mathtt{id}(\mathtt{c}) = \mathtt{i} \wedge \\
&\quad \mathtt{f}' = \mathtt{inst}(\mathtt{c}, \mathtt{n}) \wedge \\
&\quad \mathtt{S} = \mathtt{unify}(\mathtt{f}', \mathtt{f})\}
\end{aligned}
$$

$$\mathtt{I}(\mathtt{C},\mathtt{i}[\mathtt{p}_1,\ldots,\mathtt{p}_k],\mathtt{f},\mathtt{n}) = \{(\mathtt{S}_i \circ \mathtt{S},\mathtt{l}) \mid \\ \#(\mathtt{args}(\mathtt{S}(\mathtt{f}))) = \mathtt{k} \wedge \\ (\mathtt{S},\mathtt{m}) \in \mathtt{I}(\mathtt{C},\mathtt{i},\mathtt{f},\mathtt{n}) \wedge \\ (\mathtt{S}_i,\mathtt{l}) \in \mathtt{I}(\mathtt{C},[\mathtt{p}_1,\ldots,\mathtt{p}_k],\mathtt{args}(\mathtt{S}(\mathtt{f})),\mathtt{m})\}$$

$$\mathtt{I}(\mathtt{C},[\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_k],[\mathtt{f}_1,\mathtt{f}_2,\ldots,\mathtt{f}_k],\mathtt{n}) = \{(\mathtt{S}_1 \circ \mathtt{S},\mathtt{l}) \mid \\ (\mathtt{S},\mathtt{m}) \in \mathtt{I}(\mathtt{C},[\mathtt{p}_2,\ldots,\mathtt{p}_k],[\mathtt{f}_2,\ldots,\mathtt{f}_k],\mathtt{n}) \wedge \\ (\mathtt{S}_1,\mathtt{l}) \in \mathtt{I}(\mathtt{C},\mathtt{p}_1,\mathtt{S}(\mathtt{f}_1),\mathtt{m})\}$$

$$\mathtt{I}(\mathtt{C},[\,],[\,],\mathtt{n}) = \{(\varepsilon,\mathtt{n})\}$$

Im (VAR)-Fall wird ein einfacher Bezeichner `i` im Namensraum `C` nachgesehen. Ausgewählt werden aber nur *monomorphe* Namen $\mathtt{f}'$, die mit dem a-priori Namen `f` *unifizierbar* sind. Durch die Funktion `inst` kann der *monomorphe* Name $\mathtt{f}'$ Unbekannte enthalten. Für einen `k`-stellig *polymorphen* Kandidaten `c` wird der Zähler um $\mathtt{k} = \#(\mathtt{args}(\mathtt{c}))$ erhöht. Anstelle von $\mathtt{f}'$ reicht die Substitution `S` als Ergebnis.

Im (APPL)-Fall werden für eine Namensinstanz alle $\mathtt{k}+1$ Komponenten rekursiv berechnet und kombiniert. Die Reihenfolge der rekursiven Berechnungen ist im Prinzip beliebig, entscheidend ist, dass eine zuerst berechnete Substitution `S` (mit Zähler `m`) in die Folgeberechnungen einfließt; dazu wird `S` auf den a-priori Namen `f` (bzw. $\mathtt{f}_1$) angewendet und bleibt Teil des Gesamtergebnisses.

Der initiale Aufruf für eine konkrete *Funktionsdeklaration* FUN $\mathtt{f}:\mathtt{set}[\mathtt{nat},<]$ lautet (mit $\alpha = \mathtt{var}(1)$):

$$\mathtt{I}(\mathtt{S},\mathtt{set}[\mathtt{nat},<],\alpha:\text{TYPE},2)$$

6.1.1 Inferenz der Instanz

Die aktuelle Parameterliste $[\mathtt{nat},<]$ wird in Kenntnis der Struktur `Set` im nächsten Schritt wie folgt identifiziert:

$$\mathtt{I}(\mathtt{S},[\mathtt{nat},<],[\beta:\text{TYPE},\gamma:\mathtt{rel}[\beta:\text{TYPE}]],4)$$

Mit $\mathtt{inst}(\mathtt{set},2)$ wurden die formalen Parameter von `set'Set` durch *frische* Unbekannte ($\beta = \mathtt{var}(2)$ und $\gamma = \mathtt{var}(3)$) ersetzt. Ebenso wie der Typ einer Funktionsdeklaration kann ein instanziierter IMPORT $\mathtt{Set}[\mathtt{nat},<]$ identifiziert werden!

Die globale Eindeutigkeit der Strukturbezeichnung `Set` ist dabei zwar hilfreich aber nicht nötig. Tatsächlich wären *überlagerte* Strukturen für *unterschiedliche* Parameter sinnvoll, um eine durch die Parametrisierung *erzwungene* Modularisierung beim *Import* zu kompensieren!

Als Spracherweiterung wäre auch eine Einbettung von Strukturen in Module (Abschnitt 8.4) vorstellbar, bei der erst die *Qualifikation* einer Struktur mit einem global eindeutigen *Paket* eine eindeutige Strukturidentifikation liefert, falls z.B. anhand von unterschiedlichen Parametrisierungen keine Auflösung überlagerter Strukturbezeichnungen möglich ist.

6.1.2 Inferenz unbekannter Namen

In einigen Fällen, z.B. in abhängigen Parameterlisten, kann nicht nur der Typ, die Instanz und die Herkunft eines Bezeichner vervollständigt werden, sondern die gesamte Namensinstanz inferiert werden. Zur Notation solcher Namensinstanzen `inst` kann der Unterstrich „_“ verwendet werden. Der partielle Name `set`[_, <] könnte z.B allein durch die Kenntnis des Typs für die <-Relation eindeutig identifiziert werden. Bezeichnungen ohne Instanz wie `seq` oder `set`, die für eine *unbekannte Instanziierung* stehen, könnten als `seq`[_] bzw. `set`[_, _] notiert werden. Diese Notation ist präziser, weil zumindest die genaue Anzahl der Parameter angegeben wird.

Jeder Unterstrich steht für eine *frische* Unbekannte. Lediglich eine *Instanziierung* der Unbekannten, also _[...], ist nicht zulässig. (Variablen stehen nicht für einfache Bezeichner.) Die Notation _: TYPE würde aber beispielsweise nur Typnamen charakterisieren. Für die Namensidentifikation ist die Behandlung der syntaktischen Unbekannten trivial:

$$\mathtt{I}(\mathtt{C}, \_, \mathtt{f}, \mathtt{n}) = \{(\varepsilon, \mathtt{n})\}$$

Für den Unterstrich kommen alle mit dem a-priori Namen `f` kompatiblen Namen in Betracht. Die Namensinstanz des a-priori Namens ist dabei im einfachsten Fall eine Variable, die letztendlich spezialisiert werden muss oder zu einem Mehrdeutigkeitsfehler führt.

Ungünstig ist es, den ganzen Namensraum `C` bezüglich Unifizierbarkeit mit dem a-priori Namen `f` durchzuprobieren und entsprechend viele Substitutionen zurückzugeben. Das kann erstens ineffizient und zweitens für *offene* Namensräume (Abschnitt 6.3) *unvollständig* sein.

Für einen partiellen Namen `seq`[_], der nicht mit der *generischen Instanz*

aus Abschnitt 7.2.3 verwechselt werden darf, wird man nach dem *generischen* Import von `Seq` kaum einen *eindeutigen* Typ im Namensraum finden. Der Unterstrich in der Instanz `seq`[_] wäre in jedem Fall *mehrdeutig*. Konkret durch Algorithmus $\mathcal{I}$ ergibt sich der Fehler durch eine verbleibende Unbekannte (Abschnitt 6.2).

6.1.3 Identifikation von Typannotationen

Die Namensidentifikation für partielle Namen *mit expliziten Typannotationen* kann konsequent wie folgt erweitert werden:

$$\text{FUN } \mathtt{I} : \mathtt{env} \times \mathtt{name'Syn} \times \mathtt{name'Ast} \times \mathtt{nat} \to \mathtt{seq}[\mathtt{subst} \times \mathtt{nat}]$$

$$\begin{aligned}&\mathtt{I}(\mathtt{C},\mathtt{i} : \mathtt{t},\mathtt{f},\mathtt{n}) = \{(\mathtt{S_t} \circ \mathtt{S},\mathtt{l}) \mid \\ &\quad (\mathtt{S},\mathtt{m}) \in \mathtt{I}(\mathtt{C},\mathtt{i},\mathtt{f},\mathtt{n}) \wedge \\ &\quad (\mathtt{S_t},\mathtt{l}) \in \mathtt{I}(\mathtt{C},\mathtt{t},\mathtt{typename}(\mathtt{S}(\mathtt{f})),\mathtt{m})\}\end{aligned}$$

Die Ergebnisse für die Namensinstanz und den Typ können wieder wie im (APPL)-Fall in beliebiger Reihenfolge kombiniert werden. Der a-priori Name für den Typ wird durch `typename` selektiert.

Streng genommen ist eine Fallunterscheidung für Typ- und Funktionsnamen erforderlich. Für einen mit TYPE annotierten partiellen Typnamen muss auch der a-priori Name ein Typ sein. Insbesondere der partielle Name _: _ sollte nur Funktionen charakterisieren und nicht auf Typen passen:

$$\begin{aligned}&\mathtt{I}(\mathtt{C},\mathtt{i} : \text{TYPE},\mathtt{v} : \text{TYPE},\mathtt{n}) = \mathtt{I}(\mathtt{C},\mathtt{i},\mathtt{v} : \text{TYPE},\mathtt{n}) \\ &\mathtt{I}(\mathtt{C},\mathtt{i} : \text{TYPE},\_ : \_,\mathtt{n}) = \Diamond \\ &\mathtt{I}(\mathtt{C},\mathtt{i} : \mathtt{t},\_ : \text{TYPE},\mathtt{n}) = \Diamond\end{aligned}$$

6.2 Interpretation der Ergebnisliste

Eine *leere* Ergebnisliste signalisiert einen Identifizierungsfehler. Bei einer einelementigen Liste muss das Ergebnis durch die Anwendung der Substitution auf den *a-priori* Namen bestimmt werden und danach *variablenfrei* sein. Mehrelementige Ergebnislisten werden am einfachsten als *mehrdeutig* zurückgewiesen. Dieses wird genauer in Abschnitt 6.2.2 diskutiert.

6.2.1 Variablenfreiheit

Die Ergebnisliste muss nicht nur einelementig sein, das Ergebnis darf darüber hinaus *keine Variablen* enthalten. Polymorphe bzw. zu generalisierende Signaturen sind in algebraischen Sprachen nicht vorgesehen; statt implizite Variablen müssen explizit formale Parameter wie andere monomorphe Namen verwendet werden. Innerhalb der Struktur `Seq` ist also der formale Parameter α ebenso *monomorph* wie der importierte Typ `nat`. Der partielle Name `seq` wird im Namensraum $\{\forall\alpha.\texttt{seq}[\alpha]\}$, also nach einem *generischen* Import von `Seq`, nicht variablenfrei identifiziert, wohl aber im Namensraum $\{\texttt{seq}[\texttt{nat}]\}$ oder $\{\texttt{seq}[\alpha]\}$.

Eine alternative Entwurfsentscheidung wäre es, auf die Variablenfreiheit zu verzichten und verbliebene Variablen zu *generalisieren*. Auf diese Weise könnte man wie in SML *polymorphe* Funktionen innerhalb von Strukturen deklarieren. (Darüber hinaus würde man vielleicht explizite Typvariablen `'a`,`'b`, ... benutzen wollen.) Diese parametrische Polymorphie auf zwei Ebenen scheint flexibler zu sein, überlässt es aber dem Benutzer, sich für die eine oder andere Form zu entscheiden.

Die aufwendigere Notation der algebraischen Parametrisierung betrifft durch *generische Importe* (anders als in SML) nur die *Deklarationen*, nicht aber die viel häufigeren *Applikationen* polymorpher Funktionen. Ein vergleichbarer Entscheidungsspielraum besteht auch für Funktionen höherer Ordnung versus der Parametrisierung mit Funktionen (Abschnitt 5.5). Für beide Sprachaspekte, parametrische Polymorphie und Funktionen höherer Ordnung, sollte daher die *algebraische* Parametrisierung bevorzugt werden!

6.2.2 Mehrdeutigkeit

Die Namensidentifizierung des Bezeichners `seq` schlägt mit einer Mehrdeutigkeit fehl, wenn statt des generischen Namens $\forall\alpha.\texttt{seq}[\alpha]$ mehrere Instanzen, z.B. $\{\texttt{seq}[\texttt{nat}], \texttt{seq}[\alpha]\}$, im Namensraum enthalten sind. Der Identifikationsalgorithmus liefert in diesem Fall eine mehrelementige Ergebnisliste, die als Mehrdeutigkeitsfehler gewertet wird.

Ein problematischer Namensraum für die Identifizierung des partiellen Namens `seq`[`nat`] wäre:

$\{\texttt{nat}, \texttt{seq}[\texttt{nat}], \forall\alpha.\texttt{seq}[\alpha]\}$

Der generische Name *überdeckt* den instanziierten Namen und der obige Identifikationsalgorithmus liefert zwei Substitutionen, die zum gleichen vollständigen Namen führen. Kein Problem gibt es mit dem als äquivalent zu betrachtenden Namensraum, dem der instanziierte Name fehlt:

$\{\texttt{nat}, \forall\alpha.\texttt{seq}[\alpha]\}$

Ein generischer Name steht für die unbeschränkt große Menge seiner möglichen Instanzen. Insofern macht es keinen Sinn zusätzlich zu einem generischen Namen ein oder mehrere konkrete Instanzen davon in den Namensraum aufzunehmen. Solche Namensräume kann man vermeiden, wenn man vom Benutzer verlangt, Namen bzw. ganze Strukturen ausschließlich instanziiert oder generisch zu importieren. Alternativ können natürlich alle instanziiert importierten Namen ignoriert werden, sobald entsprechende Namen *generisch* importiert werden, wobei Hinweise auf irrelevante Teile im Quelltext für Benutzer durchaus nützlich wären.

Der folgende Namensraum mit *teilinstanziierten* Namen, der z.B. durch Reexporte entstehen kann, ist allerdings nicht durch einen äquivalenten Namensraum zu bereinigen:

$\{\texttt{nat}, \forall\alpha.\texttt{pair}[\alpha, \texttt{nat}], \forall\alpha.\texttt{pair}[\texttt{nat}, \alpha]\}$

Der eigentlich eindeutige partielle Name $\texttt{pair}[\texttt{nat}, \texttt{nat}]$ führt wieder zu einer mehrelementigen Liste. In OPAL (und PVS) wird daher bei einer mehrelementigen Ergebnisliste untersucht, inwieweit die unterschiedlichen Ergebnisse nach Anwendung der Substitutionen *gleich* werden. Das ist ein Zusatzaufwand, der die (theoretische) Frage nach einer insgesamt *polynomial* implementierbaren Analyse erschweren würde, da Ergebnislisten exponentiell lang werden können und damit Gleichheitstests entsprechend teuer. Einfacher ist es, *überlappende* Namensinstanzen in allen betrachteten Namensräumen auszuschließen! Dafür gibt es mehrere Möglichkeiten:

- Es dürfen nur vollständig instanziierte bzw. generische Namen direkt importiert werden. Dadurch entstehen keine *teilinstanziierten* Namen.
- Nur *überlappende* teilinstanziierte Namen sind verboten und erzwingen die Beschränkung auf *einen* Namen durch den Benutzer.
- Mehrere überlappende Namen werden *automatisch* durch einen minimalen oder den vollständig generischen Namen ersetzt.

Im letzten Fall würde der Namensraum vergrößert, was möglicherweise vom Benutzer nicht beabsichtigt oder nachvollzogen wird:

$\{\texttt{nat}, \forall \alpha\ \beta.\ \texttt{pair}[\alpha, \beta]\}$

Insgesamt kann man aber die plausible Beschränkung auf überlappungsfreie Namen Benutzern gut zumuten. Wenn aber stattdessen mehrelementige Ergebnislisten ohnehin genauer auf zusammenfallende Namen untersucht werden, dann ist es natürlich auch nicht nötig, Instanzen von generischen Namen aus dem Namensraum zu entfernen. Es ist allerdings ein (fast typischer) Fehler, den partiellen Namen `seq` als eindeutig zu identifizieren, nur weil neben dem generischen Namen zusätzlich genau eine konkrete Instanz im Namensraum vorkommt. Namensräume mit und ohne diese Instanz wären dann nicht mehr äquivalent.

6.3 Offener Namensraum

Für die Identifikation des Bezeichners `seq` im Namensraum $\{\texttt{seq}[\texttt{nat}]\}$ ist es unerheblich, ob der Typ `nat` selbst zusätzlich im Namensraum steht. Durch den (VAR)-Fall wird im Namensraum nur der Bezeichner `seq` gesucht. Lediglich wenn ein *instanzannotierter* Name wie $\texttt{seq}[\texttt{nat}]$ zu identifizieren ist, muss auch das Argument `nat` im Namensraum enthalten sein:

$\{\texttt{seq}[\texttt{nat}], \texttt{nat}\}$

Im Allgemeinen müssen also die Teile von Namen nicht selbst im Namensraum vorkommen, d.h. ein Namensraum muss *nicht abgeschlossen*, kann also *offen* sein. In PVS und OPAL mit *transitiven Importen* sowie in CASL und LPG mit *Erweiterungen* (*Extensions* bzw. *Enrichments*) sind die Namensräume *immer* abgeschlossen.

Die *Offenheit* korrespondiert zur klassischen Hindley-Milner-Typinferenz, bei der keine Typen sondern nur Funktionen in der Umgebung vorkommen. Nur *explizit* notierte Typen in Deklarationen und Annotationen müssen im Namensraum stehen. Die Vorteile offener Namensräume sind:

- Ein Namensraum kann kleiner und die Identifikation effizienter sein.
- *Untergeordnete* Namen können *versteckt* werden.
- Namen und Strukturen können *einzeln* importiert werden.

- Ein wichtiges Motiv für *transitive Importe* entfällt.

Im Gegensatz dazu sind die *selektiven* Importe von OPAL manchmal lästig. Dazu betrachten wir die folgende Struktur `A` mit einem Typ und überlagerten Funktionen `a`:

```
STRUCTURE A
  TYPE a
  FUN a: a
  IMPORT B ONLY b: TYPE
  FUN a: a → b
```

Möchte man selektiv nur die Funktion `a: a` importieren, dann erfordert der Abschluss zusätzlich den expliziten Import des Typs `a: TYPE`. Importiert man alle Namen `a`, dann muss auch der Typ `b` aus der letzten Funktionsdeklaration importiert werden. Zwar kann man `b` *transitiv* aus `A` importieren, das ist aber kein Vorteil, wenn man eigentlich von `b'B` nichts wissen möchte. Die Alternativen sind insgesamt also eher unbefriedigend:

```
IMPORT A ONLY a: a              - nicht abgeschlossen
IMPORT A ONLY a: a  a: TYPE     - aufwendig
IMPORT A ONLY a                 - nicht abgeschlossen
IMPORT A ONLY a  b              - zu viel
```

Würde man zusammen mit `a` *automatisch* auch `b` importieren, wäre der Namensraum zwar abgeschlossen, aber nicht sehr transparent. Durch *vollständige* transitive Importe – nur diese unterstützt PVS – werden Namensräume erst recht unnötig groß. Für sehr große Spezifikationen könnten dann Effizienz- und Mehrdeutigkeitsprobleme auftreten.

In offenen Namensräumen können Teile von Namen, die nicht (mehr) direkt benötigt werden, fehlen. Das verbessert die software-technisch relevante *Kapselung* (*information hiding*) von untergeordneten Details! Ein *versteckter* Name, etwa ein Typ `t`, ist *nicht sichtbar*; die Konsistenz einer Applikation `f(a)` bleibt aber garantiert. Den Argumenttyp `t` muss der Benutzer nicht kennen; er könnte auch `f(a : _)` schreiben (Abschnitt 6.1.2). Für eine explizite Typannotation `f(a: t)` kann und muss der Typ `t` *direkt* importiert werden (Kapitel 7). Im Vergleich zu Reexporten wird die direkte *Verwendung* der unsichtbaren Namen sogar verhindert!

6.4 Monotonie der Identifikation

Bei mehrelementigen Ergebnislisten darf man bei einem anschließenden Vergleich nicht den Fehler machen, Ergebnisse mit Variablen, die durch generische Namen entstehen, zu ignorieren (oder zu vergleichen). Schon ein einziges Ergebnis *mit* Variablen signalisiert eine *generische* Mehrdeutigkeit, ansonsten ginge eine für die Namensraumanalyse wichtige Monotonieeigenschaft, die auch nicht mit Verschattung vereinbar ist, verloren. Dazu betrachte man die drei wachsenden Namensräume $\mathtt{N}_1 \subset \mathtt{N}_2 \subset \mathtt{N}_3$:

$\mathtt{N}_1 = \{\mathtt{seq}[\mathtt{nat}]\}$
$\mathtt{N}_2 = \{\forall\alpha.\mathtt{seq}[\alpha]\}$
$\mathtt{N}_3 = \{\forall\alpha.\mathtt{seq}[\alpha], \mathtt{seq'Other}[\,]\}$

Der Bezeichner `seq` ist in $\mathtt{N}_1$ eindeutig `seq[nat]` und wird in $\mathtt{N}_2$ – als mehrdeutig oder nicht identifizierbar – zurückgewiesen, weil das Ergebnis nicht variablenfrei ist. Eine eindeutige Identifizierung in $\mathtt{N}_3$ als `seq'Other[]`, weil man den (unbrauchbaren) generischen Namen ausschließt oder von einer Verschattung ausgeht, widerspricht der hier propagierten *monotonen* Interpretation:

- Ein partieller Name wird durch einen Namensraum entweder gar nicht, eindeutig oder mehrfach *überdeckt* und kann auf keinen Fall durch einen *größeren* Namensraum weniger überdeckt werden.
- Ein *nicht identifizierbarer* partieller Name kann in einem kleineren Namensraum erst recht nicht identifiziert werden bzw. ein *mehrdeutiger* partieller Name muss in einem größeren Namensraum erst recht mehrdeutig sein.

Wenn `seq` im Namensraum $\mathtt{N}_2$ *mehrdeutig* ist, dann erst recht im umfassenden $\mathtt{N}_3$; ist aber `seq` in $\mathtt{N}_2$ *nicht identifizierbar*, dann erst recht im Teilraum $\mathtt{N}_1$.

Beschreibt $\mathtt{I}(\mathtt{N},\mathtt{p})$ die potenziell unendliche Menge von *variablenfreien* Namen `n`, die auf den partiellen Namen `p` im Namensraum `N` passen, dann gilt:

$\mathtt{n} \in \mathtt{I}(\mathtt{N},\mathtt{p}) \;\Leftrightarrow\; (\epsilon, \_) \in \mathtt{I}(\mathtt{N},\mathtt{p},\mathtt{n},1)$ - *Identifizierbarkeit*

$\mathtt{N}_1 \subset \mathtt{N}_2 \;\Leftrightarrow\; \mathtt{I}(\mathtt{N}_1,\mathtt{p}) \subset \mathtt{I}(\mathtt{N}_2,\mathtt{p})$ - *Monotonie*

Die Identifikation $\mathcal{I}$ induziert den Vergleich von Namensräumen und zwar so, dass generische Namen immer *alle Instanzen* umfassen.

Kapitel 7

Import

Bevor mit Hilfe der Namensidentifikation $\mathcal{I}$ aus Kapitel 6 die inkrementelle Namensraumanalyse in Kapitel 8 behandelt wird, werden hier zunächst die syntaktischen Sprachmittel für die Deklaration von Namensräumen erörtert. Außer Typ- und Funktionsdeklarationen tragen hauptsächlich *Importe* zum Namensraum bei. Importe können bezüglich der folgenden drei Dimensionen unterschieden werden:

- generisch oder instanziiert
- selektiv oder vollständig
- direkt oder transitiv

Unparametrisierte Strukturen sind Spezialfälle von parametrisierten, dabei fallen *uninstanziierte* und instanziierte Namen zusammen. Um generische Importe von parametrisierten Strukturen in Analogie zu instanziierten Importen zu behandeln, wird eine *generische Importinstanz* (Abschnitt 7.2.3) konzipiert. Die einfachsten *selektiven* Importe sind von der Form:

IMPORT $\texttt{Ide}[\texttt{name}_1, \ldots, \texttt{name}_k]$ ONLY `ide`

`Ide`[1] ist eine *bekannte* Struktur, die in OPAL eindeutig bezeichnet ist und aus der alle exportierten überlagerten Namen mit dem Bezeichner `ide` zum Namensraum der importierenden Struktur hinzukommen. Die partiellen Namen $\texttt{name}_i$ $(1 \leq i \leq k)$ in der Importinstanz sind zu identifizierende *aktuelle* Parameter für die Struktur `Ide`.

[1] Die Großschreibweise für Strukturen ist lediglich eine Namenskonvention.

Beim *generischen* Import, wie ihn OPAL und PVS unterstützen, wird syntaktisch die Instanz weggelassen. Als Notationsalternative wird hier eine Visualisierung der *generischen Instanz* vorgeschlagen (Abschnitt 7.2.3). Die Anzahl der Unterstriche muss dabei identisch zur Anzahl `k` der formalen Parameter von `Ide` sein.

```
IMPORT Ide ONLY ide            - oder
IMPORT Ide[_, ..., _] ONLY ide
```

Ohne Namensidentifikationen können die Namen mit dem Bezeichner `ide` durch eine einfache Filteroperation dem Namensraum der Struktur `Ide` entnommen werden. Die Struktur `Ide` wird dabei als unparametrisiert und *ohne formale Parameter* betrachtet. Formale Parameter werden nicht exportiert, sie sind aber i.d.R. Teilkomponenten der exportierten Namen. Beim generischen Import werden daher die freien Vorkommen von formalen Parametervariablen – individuell für jeden importieren Namen `ide` – *gebunden*. (Alle Namensräume sind endliche Listen von vollständigen Namen mit oder ohne gebundenen Variablen.)

Eine Neuerung sind *überlagerte* Strukturen `Ide` (Abschnitt 6.1.1). Beim *uninstanziierten* Import werden die Namen *aller* Strukturen `Ide` berücksichtigt. *Typannotierte* Platzhalter in der Importinstanz ermöglichen bei Bedarf die Auflösung der Strukturüberlagerung!

Durch *vollständigen* Import können *alle* Namen einer Struktur importiert werden, ohne die einzelnen Bezeichner aufzählen zu müssen.

```
IMPORT Ide[...] COMPLETELY
```

Ein vollständiger Import kann als Abkürzung von *mehreren* selektiven Importen betrachtet werden. Die Bezeichner $\texttt{ide}_\texttt{i}$ kann man der Struktur `Ide` entnehmen.

```
IMPORT Ide[...] ONLY ide_1  ide_2  ...   - oder

IMPORT Ide[...] ONLY ide_1
IMPORT Ide[...] ONLY ide_2
                     ...
```

Für zu importierende Strukturen muss festlegt sein, welche konkreten Namen – natürlich keine formalen Parameter – *exportiert* werden. Dafür gibt es zwei vertretbare Alternativen:

1. Es werden nur Namen exportiert, die in der Struktur *neu* als Typ oder Funktion *deklariert* wurden.

2. Es werden alle Namen exportiert, die im *Namensraum* der importierten Struktur *bekannt* sind.

Außerdem sind noch explizite Exportdeklarationen sowie die Trennung in eine *exportierte* Schnittstelle (SIGNATURE) und versteckte Implementierung wie in OPAL vorstellbar.

Die zweite für PVS-Theorien und OPAL-Schnittstellen realisierte Alternative unterstützt *Reexporte*, d.h. auch die innerhalb einer Struktur importierten Namen können *transitiv* importiert werden. Dadurch werden bei einem *vollständigen* Import alle Namen importiert, die in untergeordneten Strukturen für die dortigen Deklarationen importiert wurden, d.h. alle Namen und ihre Komponenten werden transitiv *abgeschlossen* importiert.

Bei der ersten Alternative können Namen nur *direkt* importiert werden. Dadurch ist die Herkunft und Instanz importierter Namen bekannt und es ergeben sich eine Reihe von Vereinfachungen (Abschnitt 7.3).

Im Folgenden müssen jetzt nur noch *Importinstanzen* (Abschnitt 7.1), *selektive* Importe mit Annotationen (Abschnitt 7.2) und ihre Kombinationen mit *direkten* (Abschnitt 7.3) und *transitiven* Importen (Abschnitt 7.4) erläutert werden.

7.1 Eindeutige Importinstanz

Ebenso wie der Typ von Funktionsdeklarationen monomorph und eindeutig sein muss, sollten Importinstanzen *monomorph und eindeutig* sein! Dieses ist keine Selbstverständlichkeit:

```
STRUCTURE S
  IMPORT Seq ONLY seq
  IMPORT Option[seq] ONLY option   - generisch
                          ...
  FUN f : seq                      - generisch und illegal
```

Die Verwendung des generischen Namens `seq` in der Instanz `Option[seq]` ist in OPAL illegal. In PVS wird aber einfach folgender *teilinstanziierter* Name importiert:

$\forall\alpha$.`option'Option`[`seq'Seq`[α]]

In Abschnitt 7.1.2 wird begründet, warum man derartige generischen Instanzen ebenso wie Namen mit Variablen (Abschnitt 6.2) als *mehrdeutig* zurückweisen sollte. Für konkrete *überlagerte* Instanzen, d.h. nur endlich statt unendlich vieler Instanzen, wird in OPAL *und* PVS eine *Mehrdeutigkeit* diagnostiziert:

```
...
IMPORT Seq[nat] ONLY seq
IMPORT Seq[real] ONLY seq
IMPORT Option[seq] ONLY option    - mehrdeutige Instanz
```

Bei mehrdeutigen Instanzen könnte man theoretisch entsprechend mehr importieren, diese Alternative wird in Abschnitt 7.1.2 diskutiert:

```
option'Option[seq'Seq[nat]]
option'Option[seq'Seq[real]]
```

Mehrdeutige Instanzen bleiben unabhängig von *Applikationen* importierter Namen mehrdeutig:

```
IMPORT A ONLY t
IMPORT B ONLY t
IMPORT Seq[t] ONLY seq    - mehrdeutige Instanz
FUN f : seq[t'A]
```

Die Funktionsdeklaration trägt nicht zur Eindeutigkeit des Imports bei (siehe Abschnitt 7.1.1). Meistens ebenso kurz und besser lesbar wäre stattdessen die Annotation beim Import:

```
IMPORT A ONLY t
IMPORT B ONLY t
IMPORT Seq[t'A] ONLY seq    - eindeutig
FUN f : seq
```

Die Importinstanz [`t'A`] ist nicht nur eindeutig und monomorph, sondern *lokal* und unabhängig von der Funktionsdeklaration für `f` auflösbar.

Mehrdeutiger Zyklus

Durch einen gemeinsamen Namensraum für alle Importinstanzen, wie dieser in OPAL durch von der Reihenfolge unabhängige Deklarationen entsteht, verdient noch das folgende *zyklische* Beispiel Beachtung:

```
IMPORT Nat ONLY nat
IMPORT Seq[nat] ONLY seq
FUN f : seq                    - mehrdeutig
IMPORT Seq[seq] ONLY seq       - mehrdeutig
```

In PVS würden auf Grund der linearen Sichtbarkeit die beiden monomorphen Namen `seq[nat]` und `seq[seq[nat]]` importiert und `f` mit Typ `seq[nat]` deklariert. In OPAL ist `f : seq` *mehrdeutig* ebenso wie in PVS, wenn `f : seq` *hinter* dem letzten Import stehen würde (Kapitel 8).

In OPAL ist auch die Instanz `Seq[seq]` mehrdeutig, obwohl man die zweite Vervollständigung `Seq[seq[seq[nat]]]` ignorieren könnte, weil dadurch noch ein dritter Name `seq[seq[seq[nat]]]` hinzu käme. Eine zyklische Interpretation des Imports würde zu einer unendlichen Kette monomorpher Namen führen, die für eine Repräsentation mit gebundenen Variablen nicht mehr geeignet ist. Eine Namensmenge der Form $\texttt{seq}^{+}\texttt{[nat]}$ wäre ein Problem von neuer Qualität! (Abschnitt 7.1.2)

Insgesamt wird eine *lokale* Auflösbarkeit von Importinstanzen verlangt und diese ist die Grundlage für die inkrementelle Namensraumanalyse aus Kapitel 8. Praktisch ist die lokale Auflösbarkeit genau das, was man als Benutzer erwartet und nachvollziehen kann. Die Mehrdeutigkeit für `Seq[seq]` ist konform mit einer Mehrdeutigkeit für eine Funktionsdeklaration FUN `f : seq`. Für beide Fälle erfolgt dieselbe Namensidentifikation (Kapitel 6).

Die Idee einer Namensvervollständigung durch eine globale Suche [DGMP97] legt aber auch eine *globale* Auflösbarkeit nahe, die im folgenden Abschnitt 7.1.1 diskutiert wird. Der spezielle Nutzen ist aber so gering, dass man über eine Realisierung nicht nachdenken muss.

7.1.1 Lokale versus globale Auflösbarkeit

Wie beim mehrdeutig instanziierten Import von `Seq[t]` angedeutet, trägt eine nachfolgende *Applikation* `seq[t'A]` nicht zur Auflösung der Mehrdeutigkeit bei, obwohl *global* betrachtet die eindeutige Interpretation von `Seq[t]` als

`Seq[t'A]` möglich ist, da die andere Auflösung `Seq[t'B]` zum Fehler in der Applikation `seq[t'A]` führen würde.

Übertragen auf Ausdrücke entspricht dieses der Typanalyse von LET-Variablen aus Kapitel 4. Die Überlagerungsauflösung einer LET-Variablen erfolgt *global* in Abhängigkeit von den *Applikationsstellen* im *nachfolgenden* IN-Ausdruck. Analog ist die lokale Auflösbarkeit für Importinstanzen mit der lokalen LET-Restriktion (aus Abschnitt 4.2) vergleichbar, die in PVS durch λ-Variablen *mit Typ* realisiert ist.

Eine *globale* Typanalyse ist aber leichter, wenn dafür der Namensraum schon bekannt ist. Bei der Signaturanalyse ist der Namensraum gerade erst zu ermitteln. Dieses *Henne-Ei-Problem* ist aber allein durch die Analogie zwischen der Namens- und Ausdrucksebene *eindeutig lösbar:*

$$\text{globale Signaturanalyse} \quad \cong \quad \text{LETREC-Typanalyse}$$

Wie Typen für wechselseitig rekursive Funktionen mit Überlagerungen müssen die Funktionsdeklarationen und Importinstanzen aufgelöst werden. Dafür müssen analog zu $\mathcal{W}_o$ Unbekannte in der Umgebung für $\mathcal{I}$ verwaltet werden. Dieses ist so in der existierenden Signaturanalyse von OPAL angelegt: aber *Applikationsstellen* werden nicht für Spezialisierungen der Unbekannten berücksichtigt und deswegen wird nur lokal aufgelöst. Erschwerend für OPAL sind zudem partielle ONLY-Namen (Abschnitt 7.4) und Wiederholungen (Abschnitt 8.1.3), die durch obige Analogie nicht erfasst werden.

Ebenso wie auf *vernünftige Deklarationen* kann man auch auf *lokale Auflösbarkeit* von Namen bestehen. Je globaler die Abhängigkeiten sind, die tatsächlich für die Identifikation oder Überlagerungsauflösung benötigt werden, desto schwerer sind sie auch vom Benutzer nachvollziehbar und wartbar. Indiskutabel wäre es natürlich, mit diesem Argument ganz auf Überlagerung zu verzichten. Der Nutzen von Überlagerung ist in der Praxis beträchtlich! Über graduelle Beschränkungen der Überlagerung und entsprechende Algorithmen (siehe auch Kapitel 3 oder die Abschnitte 5.4.2 und 5.4.3) mag man geteilter Meinung sein.

7.1.2 Generische versus mehrfach instanziierte Namen

Statt generische Namen oder überlagerte Namen innerhalb von Importinstanzen `Option[seq]` als mehrdeutig zurückzuweisen, kann man alternativ *mehr*

importieren. Das ist bei einer endlichen Überlagerung von `seq` kein Problem und generische Namen können wie eine spezielle Instanz (siehe Abschnitt 7.2.3) behandelt werden.

Aus Monotoniegründen, z.B. wenn man viele instanziierte Importe durch einen generischen ersetzt, ist es sinnvoll (Abschnitt 6.4), generische Namen im Vergleich zu überlagerten Namen erst recht als mehrdeutig zurückzuweisen! Im genauen Gegensatz zu PVS könnte man daher besser nur endliche Überlagerungen als generische Namen in Instanzen akzeptieren. Dieses wurde von Reuleaux [Reu98] implementiert. Ohne die lineare Sichtbarkeit ergab sich allerdings für obiges zyklische Beispiel die Namensmenge $\texttt{seq}^+[\texttt{nat}]$ durch eine Endlosschleife:

```
IMPORT Seq[seq] ONLY seq    - endlos
IMPORT Seq[nat] ONLY seq
```

Ignoriert man dieses Problem, dann kann der Namensraum noch explodieren. Für den (implizit generisch importierten) `n`-stelligen Tupeltypkonstruktor ergeben sich 2^n Instanzen für `seq`:

```
IMPORT A ONLY t
IMPORT B ONLY t
IMPORT Seq[t × ... × t] ONLY seq
```

Zusammen mit den generischen PVS-Instanziierungen ergibt sich eine Analogie zu den um Überlagerung verallgemeinerten *polymorphen* LET-Ausdrücken (Abschnitt 4.3), auf die man – orthogonalerweise – in algebraischen Sprachen zu Gunsten *monomorpher* LET-Ausdrücke verzichtet.

Der Ausdruck LET `f` = ◊ ist nach einem generischen Import *polymorph* korrekt. Nach mehreren *instanziierten* Importen von ◊'`Seq` liegt aber *Überlagerung* vor, die für „echte“ Überlagerung zu LET-Variablen führt, die nicht mehr *uniform* definiert sind.

Die Analogie zu den üblichen polymorphen LET-Ausdrücken ohne Überlagerung legt genau die für PVS getroffene Entwurfsentscheidung für generische Instanziierungen nahe. Konsequenter ist die für OPAL gültige Analogie zu *monomorphen* LET-Ausdrücken. Unabhängig davon ist der Nutzen generisch instanziierter Importe begrenzt. Das Ziel, *teilinstanziierte* Namen zu bilden, wird nur sehr umständlich erreicht:

```
IMPORT Nat ONLY nat
IMPORT Option ONLY option          - generisch
IMPORT Pair[option, nat] ONLY pair
```

Der teilinstanziierte Name $\forall\alpha.$`seq'Seq`[`seq'Seq`[α]] kann *nicht* gebildet werden. Durch einen uninstanziierten Import von `seq'Seq` wäre der teilinstanziierte Name schon überdeckt und weitere Importe von `seq'Seq` wären nutzlos:

```
IMPORT Seq ONLY seq
IMPORT Seq[seq] ONLY seq
```

Für teilinstanziierte Namen fehlen also noch Sprachkonzepte. Zudem muss die Überlappungsproblematik für Identifikationen behandelt werden (Abschnitt 6.2.2). Eleganter wäre, generische Namen mit dem Platzhalter für formale Parameter zu notieren (Abschnitt 7.2.3):

```
IMPORT Pair[_, nat] ONLY pair
```

Teilinstanziierte Namen können auch durch Reexport (Abschnitt 7.4.4) oder Synonyme (Abschnitt 8.4.3) entstehen.

7.2 Selektiver Import

Ein Motiv für selektive Importe ist, den Namensraum möglichst klein und überschaubar zu halten. Durch vollständige Importe, für die es in PVS keine Alternativen gibt, wird manchmal zu viel importiert. Statt aber wie in OPAL die erwünschten Namen aufzulisten, wird zunächst in Abschnitt 7.2.1 vorgeschlagen, unerwünschte Namen auszuschließen.

Beim selektiven Import mit dem Bezeichner `ide` nach ONLY werden alle *überlagerten* Namen importiert. Wieviele bzw. welche der überlagerten Namen importiert werden, kann zwar vom Compiler genau gemeldet werden, ist aber besonders bei Reexporten für den Benutzer nicht sehr transparent. Um ausgewählte überlagerte Namen auch einzeln importieren zu können, darf der Bezeichner `ide` *annotiert* werden. Konkret kann in OPAL nach ONLY eine *Liste partieller Namen* angegeben werden.

Um Namensidentifizierungen zu erleichtern, wird in Abschnitt 7.2.2 auf einen vorteilhaften *getrennten* Import von Typen und Funktionen verwiesen. Aus generischen Importen können Namen direkt nur generisch importiert werden.

Um illegale Instanziierungen durch Instanzannotationen auszuschließen, wird in Abschnitt 7.2.3 eine spezielle *generische Instanz* aus Platzhaltern für formale Parameter konzipiert. Diese Platzhalter ermöglichen vor allem *exakte Typannotationen.*

Für *direkte* Importe (Abschnitt 7.3) ergeben sich wesentliche Erleichterungen zur Festlegung von ONLY-Namen. Beim Reexport werden Herkunfts- und Instanzannotationen für zu importierende Namen berücksichtigt. Sämtliche Probleme für transitive Importe enthält Abschnitt 7.4.

7.2.1 Selektiver Ausschluss

Ein spezieller Grund für selektive Importe kann darin liegen, *unerwünschte Überlagerungen* zu vermeiden, um sich z.B. Herkunftsannotationen für typgleiche Funktionen zu ersparen. Beim Import aus zwei Strukturen würde ein überlagerter Bezeichner dazu in einer ONLY-Liste einfach nicht aufgeführt. Die explizite Auflistung aller anderen (nötigen oder nicht störenden) Bezeichner könnte allerdings mühsam sein. Komplementär zu den ONLY-Importen wären daher zusätzliche ALLBUT-Importe sinnvoll:

```
IMPORT Nat ALLBUT +
IMPORT Real ONLY + ...
```

Ein Problem beim Ausschluss von Namen ist es, wenn derselbe Name an mehreren Stellen importiert wird. Vollständige und insbesondere transitive Importe könnten einen sorgfältigen Namensausschluss wieder zunichte machen. Vollständige und selektive ALLBUT- und auch ONLY Importe derselben Instanz einer Struktur machen *zusammen* also nicht viel Sinn. Für Namen, die durch ALLBUT explizit nicht importiert werden sollen, will man zumindest wissen, ob sie nicht doch durch andere Importe hinzukommen. Importe sollten sich also nicht widersprechen.

Bei direkten Importen (Abschnitt 7.3) können Kollisionen schon syntaktisch vermieden werden, indem man pro Instanz einer Struktur entweder

- einen COMPLETELY- oder
- einen ALLBUT-Import oder
- nur ONLY-Importe erlaubt.

Davon unberührt kann die Möglichkeit bleiben, dieselben Importe zu *wiederholen*.

Der *Ausschluss spezieller Instanzen* würde die Repräsentation der Namensräume erschweren:

IMPORT `Seq` ALLBUT `seq[bool]` - *illegal*

Da aber auch der entsprechende ONLY-Import problematisch ist, sollte man eine *Instanzannotation* für den importierten Bezeichner gar nicht vorsehen (Abschnitte 7.2.3 und 7.4.2).

Ein Nachteil von ALLBUT-Importen ist, dass die verwendbaren Bezeichner der importierten Namen nicht offensichtlich sind. Das gilt auch für vollständige Importe aber verschärft für ALLBUT, weil *nicht-verwendbare* Bezeichner explizit aufgeführt werden können. Für *überlagerte* Bezeichner ist aber ein selektiver Ausschluss ideal, ansonsten sind ONLY-Importe vorzuziehen.

7.2.2 Trennung von Typen und Funktionen

Die Gleichberechtigung von Typen und Funktionen ist bezüglich der Parametrisierung und Instanziierung beabsichtigt; der Benutzer muss allerdings Typen und Funktionen gut unterscheiden können. Funktionen werden – momentan – hauptsächlich in Ausdrücken und – im Vergleich zu Typen seltener – in Instanzen appliziert. Beim Import eines unannotierten Bezeichners `ide` ist diese Unterscheidung syntaktisch nicht so offensichtlich wie bei Typ- und Funktionsdeklarationen. Typannotationen für Funktionen oder (wiederholte) TYPE-Annotationen für (mehrere) Typen sind umständlich und deswegen sollte ein syntaktisch getrennter Import von Typen und Funktionen erwogen werden, etwa in der folgenden Form:

IMPORT `Seq[nat]` ONLY TYPE `seq`
FUN :: ◊ ...

Überlagerungen zwischen Typen und Funktionen wären auf diese Weise leichter kontrollier- und erkennbar. Der selektive Import von Funktionen, ohne überlagerte *Typen* mit zu importieren, die gar nichts mit der Funktion zu tun haben, erfordert *Typannotationen*, die man lieber nur zur Trennung überlagerter *Funktionen* verwenden würde.

7.2.3 Generische Instanz

Die Typannotation von instanziiert importierten Funktionen erfolgt mit ihren *instanziierten* Typen. Wie sollen aber Typannotationen für importierte Funktionen angegeben werden, wenn die formalen Parameter im generischen (oder im zu konstruierenden) Namensraum nicht zur Verfügung stehen? In OPAL sind die Typannotationen entsprechend beschränkt:

IMPORT `Seq` ONLY `seq`: TYPE $\Diamond$: `seq` - *legal*
IMPORT `Seq` ONLY `ft`: `seq` $\rightarrow \alpha$ - *illegal*

Dadurch lassen sich insbesondere die überlagerten Funktionen `seq` vom Typ $\alpha \rightarrow$ `seq` und $\alpha \times \alpha \rightarrow$ `seq` (aus Abschnitt 5.4.3) nicht getrennt generisch importieren.

Die Angabe einer *instanziierten* Typannotation ist – ebenso wie die direkte Instanzannotation – in OPAL für generische Funktionsnamen nicht zulässig; lediglich *partielle* Typnamen, in denen formale Parameter weggelassen werden, sind für Typannotationen geeignet. Aber selbst der (in OPAL unzulässige) partielle Name _ $\rightarrow$ `seq` würde sowohl auf den Typ $\alpha \times \alpha \rightarrow$ `seq` als auch auf $\alpha \rightarrow$ `seq` passen. (Nur _ $\times$ _ $\rightarrow$ `seq` würde nicht auf $\alpha \rightarrow$ `seq` passen.)

Wenn man bei generischen Importen – wie üblich – vollständig von den Namen formaler Parameter abstrahieren und trotzdem jede Funktionalität annotieren können möchte, dann ist das elegant mit einem extra *Platzhalter* für formale Parameter möglich. Interessanterweise kann der Platzhalter für alle Parameter *gleich* gewählt werden, da Funktionalitäten von überlagerten Funktionen einer Struktur für keine Instanz zusammenfallen (Abschnitt 5.4.2), also auch nicht für eine *hypothetische* Instanz aktueller Parameter mit *gleichen Namensinstanzen*!

Als Syntax für diesen Platzhalter wird *wieder* der Unterstrich gewählt, der damit seine Bedeutung als *beliebige* Namensinstanz (Abschnitt 6.1.2) zumindest für generische Importe einbüßt. (Denkbar wäre es auch, ein anderes Schlüsselsymbol für die formalen Parameter zu wählen. Eine beliebige Namensinstanz ist allgemeiner als ein formaler Parameter und sollte daher länger sein; z.B. wären auch zwei Unterstriche für Namensinstanzen bzw. ein Punkt für formale Parameter geeignet.)

Die Typannotation _ $\rightarrow$ `seq` wird nun mit der Interpretation des Unterstrichs als formaler Parameter eindeutig und jede generische Funktionalität kann so bei Bedarf immer eindeutig notiert werden. Instanzannotationen innerhalb der Typen sind natürlich erlaubt:

```
IMPORT Seq ONLY ft: seq → _
IMPORT Set ONLY # : set[_, _ : rel[_]] → nat
```

Die in den Typannotationen verwendeten Namen müssen natürlich *nicht* importiert werden. Diese Namen sind im Namensraum der importierten Struktur bekannt und dienen lediglich zur Identifizierung des tatsächlich zu importierenden Namens.

Die Notation `Set[_, _]` entspricht einer hypothetischen Instanz, während semantisch die importierten Namen bezüglich der formalen Parameter polymorph bleiben. Der Versuch einer (durch Typannotationen indirekten) Instanziierung mit *monomorphen* Namen, die sich vom Platzhalter für die formalen Parameter unterscheiden, würde zu einem Identifizierungsfehler führen. Generische Namen werden immer mit ihrer *generischen Instanz* importiert.

Um *überlagerte* Strukturen generisch getrennt zu importieren, kann man *typannotierte* Platzhalter oder nur die Typen in der Importinstanz angeben:

```
IMPORT Set[_ : TYPE, _ : rel[_]] ONLY set    - oder
IMPORT Set[TYPE, rel[_]] ONLY set
```

Eine Instanz `S[TYPE, _ : _]` korrespondiert zur Parametrisierung $\mathtt{S}[\alpha, \beta : \alpha]$. Welche Parametrisierung passt zu `T[TYPE, _]`? Steht also der Platzhalter für die *Namensinstanz* des zweiten Parameters oder für seinen Typ? Das kann man beliebig festlegen oder Eindeutigkeit per *Doppelpunkt* erreichen: `T[TYPE, : _]` oder `T[TYPE, _ : ]`. Ganz allgemein ist ein führender oder folgender Doppelpunkt eine geeignete und *minimale Annotation*, um Typen und Funktionen zu unterscheiden!

7.3 Direkter Import

Der *direkte* instanziierte selektive Import entspricht im Prinzip der Angabe einer *Namensinstanz* (Kapitel 5). Da die Herkunft und Instanz feststeht, können überlagerte Namen `ide` nur *typverschieden* sein (Abschnitt 5.4.2) und eine Typannotation wäre ausreichend, um einen vollen *Namen* zu erhalten:

$\mathtt{ide'Ide}[\mathtt{name}_1, \ldots, \mathtt{name}_k]$

Die Namen $\texttt{name}_\texttt{i}$ innerhalb der *Instanzen* stammen aus *anderen* Importen oder Deklarationen; ansonsten ergäbe sich ein zyklischer Name. *Generisch* importierte Namen `ide` sind entsprechend der formalen Parameterliste der Struktur `Ide` gleichartig polymorph:

$\forall\alpha_1\ \ldots\ \alpha_\texttt{k}.\ \texttt{ide'Ide}[\alpha_1, \ldots, \alpha_\texttt{k}]$

Beim direkten Import ist die Menge der potenziell überlagerten Namen stark beschränkt; es ist daher keine große zusätzliche Beschränkung, wenn man eine *eindeutige* Identifizierung von *typannotierten* Bezeichnern fordert. Damit kann die *unmodifizierte* Ergebnisinterpretation der Namensidentifikation (aus Abschnitt 6.2) wiederverwendet werden!

Typannotationen für Bezeichner sind ausreichend und redundante andere Annotationen bieten keine Vorteile. Scheinbar zyklische Importe sind nicht möglich:

```
IMPORT Seq[nat : TYPE] ONLY nat : TYPE
IMPORT Seq[real : TYPE] ONLY real : TYPE
IMPORT Seq[seq : TYPE] ONLY seq : TYPE
```

Die Namen `nat'Nat` (per Reexport) und `real'Real` mit unpassender Herkunft können offensichtlich nicht aus `Seq` importiert werden. Wenn aber im letzten Fall `seq'Seq : TYPE` importiert wird, dann ergäbe sich entweder ein illegaler zyklischer Name oder für einen *anderen* Namen `seq : TYPE` mit einer *anderen Herkunft* eine Mehrdeutigkeit an der Instanz `Seq[seq : TYPE ]`! (vgl. Abschnitt 7.1.2)

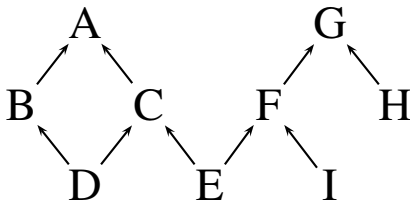
Wie in Abschnitt 7.2.1 ausgeführt, ist der selektive Ausschluss *direkt* viel besser kontrollierbar. Ein getrennter Import für Typen und Funktionen (Abschnitt 7.2.2) ist aber immer noch sinnvoll. Zwar kann es in einer importierten Struktur `Ide` maximal einen Typ mit dem Bezeichner `ide` gegeben, dennoch sollte man diesen nicht zusammen mit überlagerten Funktionen `ide` importieren müssen. Lange Typannotationen für Funktionen (nach ONLY), nur um den Import eines überlagerten Typs auszuschließen, möchte man natürlich vermeiden.

Direkte Importe sind nur dann akzeptabel, wenn oder weil der Namensraum bezüglich der Typen *offen* sein kann (Abschnitt 6.3). Da die Importinstanz im Namensraum identifizierbar sein muss, sind *alle* Namen immer *bezüglich ihrer Instanzen abgeschlossen!* Vor allem vermeiden aber direkte Importe die folgenden Nachteile transitiver Importe:

- Unterschied zwischen Import- und Herkunftsstruktur (Abschnitt 7.4.1)
- Identifikation partieller Namen nach ONLY (Abschnitt 7.4.2)
- Import generischer Namen aus instanziierten Strukturen (7.4.3)
- Import von instanziierten Namen aus generischen Strukturen (7.4.4)
- Namensraumexplosion (Abschnitt 7.4.5)
- seltene globale Instanzabhängigkeiten (Abschnitt 7.4.6)
- Vielfachimporte transitiver Namen.

7.4 Transitiver Import

Die Importrelation von algebraischer Strukturen ist *gerichtet* und *azyklisch*. Eine graphische Darstellung ist beispielsweise:



Die Struktur `A` importiert die Strukturen `B` und `C` *direkt* sowie `D` und `E` *transitiv*. Wie die untergeordneten Strukturen `B`, `C`, `D` und `E` konkret instanziiert sind oder welche Namen daraus in der Toplevel-Struktur `A` verwendbar sind, ist dieser Relation nicht zu entnehmen. Der *DAG* kann *topologisch sortiert* werden und bestimmt dadurch eine *Analysereihenfolge* (Kapitel 8). Untergeordnete Strukturen sind vollkommen *unabhängig* von übergeordneten.

Durch ausschließlich *vollständige* und *generische* Importe stehen immer *alle* Namen transitiv abgeschlossen zur Verfügung. Durch *instanziierte* Importe von `D` innerhalb von `B` und `C` wird *transitiv* die *Summe der Instanzen* für `A` exportiert. Durch *selektive* Importe von `D`, werden entsprechend über `B` und `C` die selektierten Namen für `A` reexportiert.

Andere Instanzen von oder Namen aus `D` müssen *direkt* aus `D` oder transitiv über andere Strukturen importiert werden. Aus der Abgeschlossenheit der

Namen in `B` und `C` folgt die Abgeschlossenheit für `A`; da durch mehrere Instanzen höchstens Namen zusammenfallen können. Um diese Abgeschlossenheit immer zu gewährleisten, müssen *selektiv* importierte Namen *abgeschlossen* sein (Abschnitt 6.3). *Deklarierte* Namen sind per Konstruktion abgeschlossen.

Zwei Vorteile transitiver Importe entsprechen insofern denen von Modularisierungskonzepten (Abschnitt 8.4):

1. Man kann *viele* Namen aus *einer* Struktur importieren.
2. Die genaue *Herkunft* eines importierten Namens kann *versteckt* bleiben.

Um den ersten Vorteil für *selektive* Importe auszunutzen, ist eine Syntax für *mehrere* ONLY-Namen wichtig. Der Nachteil, manche Namen nur zwecks Abgeschlossenheit importieren zu müssen, wird dadurch gemildert:

IMPORT `Ide`[...] ONLY $\texttt{name}_1$ $\texttt{name}_2$ ...

Ebenso wie für einen ONLY-Bezeichner `ide` *mehrere* überlagerte Namen importiert werden, muss auch ein *partieller* Name $\texttt{name}_\texttt{i}$ nicht eindeutig sein. Alle auf $\texttt{name}_\texttt{i}$ *passenden* Namen werden importiert. Konkret in OPAL ist ein partieller Name ein *annotierter* Bezeichner. Mit dem Platzhalter aus Abschnitt 6.1.2 hätte man eine universellere *Maske*.

Beim Reexport müssen die Herkunft und die Instanz importierter Namen nicht der importierten Struktur entsprechen. Um transitiv importierte Namen genauer zu selektieren sind Herkunfts- und Instanzannotationen erlaubt:

```
IMPORT Nat ONLY <                  - direkt
IMPORT Seq ONLY nat'Nat            - transitiv
IMPORT Set[nat, <] ONLY seq[nat]   - transitiv
```

Sinnvollere Beispiele ergeben sich erst für überlagerte Namen. Notwendige Herkunftsannotationen für überlagerte Namen unterminieren das Verstecken der Herkunft (Abschnitt 7.4.1). Unintuitiv ist die Kluft von Herkunfts- und Instanzannotationen im Vergleich zur Importinstanz. Erlaubt man wie beim direkten Import nur Typannotationen können einige Überlagerungen nicht gut vermieden werden und dann kann man fast wie in PVS ganz auf *selektive* transitive Importe verzichten. In Verbindung mit der Abgeschlossenheit gilt dieses Argument erst recht!

Wie beim direkten Import, der keine Instanzannotation unterstützt, kann ein *generischer* Name aus einer uninstanziierten Struktur *nicht instanziiert* importiert werden:

IMPORT `Seq` ONLY `seq[nat]` - *illegal*

Diese Restriktion wird in Abschnitt 7.4.2 begründet. Für transitiv importierte Namen ergeben sich dadurch Probleme für die Kombination von generischen und instanziierten Importen (Abschnitt 7.4.3). Durch diese Kombination können Namen auch *teilinstanziiert* importiert werden (Abschnitt 7.4.4). Verzichtet man auf die Mischung von generischen und instanziierten Importen, dann können sich durch instanziierte transitive Importe theoretisch exponentiell viele Namen ergeben (Abschnitt 7.4.5). Außerdem können transitiv importierte Namen aktuelle Parameter in der Importinstanz sein, die im schlimmsten Fall nicht mehr lokal auflösbar sind (Abschnitt 7.4.6).

In letzter Konsequenz verbleiben lediglich *vollständige generische transitive* Importe. Für eine transitiv importierte Struktur `D` lässt sich der *mehrfache* Import über `B` und `C` nach `A` leicht behandeln.

7.4.1 Herkunftsannotation

Herkunftsannotationen erlauben zwar Überlagerungen zu vermeiden, sind aber verwirrend:

IMPORT `A` ONLY `ide'B`
IMPORT `A` ONLY `ide'A`

Im ersten Fall fragt man sich, warum nicht direkt aus `B` importiert wird und im zweiten, warum man den Strukturbezeichner `A` doppelt angeben muss. Die erste Frage ist (noch) berechtigt. Im zweiten Fall will man einen überlagerten Namen `ide'C` oder `ide'D` *nicht* importieren. Im Vergleich dazu betrachte man die Alternative:

IMPORT `A` ONLY `ide'A ide'B`

Syntaktisch spart man den IMPORT `B`, wobei noch offen ist, ob dann ein ONLY-Bezeichner `ide` reicht oder `ide'B` selektiert werden muss. Wenn aber ein überlagerter Name `ide'C` tatsächlich ein Problem für die Formulierung von Ausdrücken ist und deswegen ausgeschlossen werden soll, dann steht man vor einem Dilemma: entweder man annotiert beim Import oder später am Ausdruck. In beiden Fällen ist die *versteckte* Herkunft kein Vorteil.

Um zur Vermeidung von Mehrdeutigkeiten eine versteckte Herkunft nicht offenbaren zu müssen, wurde erwogen, statt einer Herkunftsannotation auch eine *Importannotation* zuzulassen:

```
STRUCTURE S
  TYPE t

STRUCTURE T
  IMPORT S ONLY t
  TYPE t
```

Beim Import der Struktur `T` sind durch Reexport die Namen `t'S` und `t'T` im Namensraum. Sind Import- und echte Herkunftsannotationen ununterscheidbar, dann wäre die Notation `t'T` mehrdeutig, es sei denn der Name `t` *aus* `T` *verschattet* `t'S`.

Es besteht ein Unterschied zwischen der eigentlichen *Herkunft* und dem *Herkommen*. Ersteres ist *die* Struktur, in der ein Name *deklariert* wird; letzteres ist irgendeine Struktur, aus der der Name transitiv *importiert* werden kann.

Wegen der Probleme bei Überlagerungen kann man auf Herkunfts-, Import- und auch Instanzannotationen in ONLY-Namen besser ganz verzichten.

7.4.2 Instanzannotation

Beim obigen illegalen Import von `seq[nat]` aus der generischen Struktur `Seq` fragt man sich, warum das nicht genau wie ein instanziierter Import aus `Seq[nat]` mit dem ONLY-Bezeichner `seq` behandelt wird. Der Unterschied besteht in der Identifizierung der Importinstanz (Abschnitt 7.1) und dem partiellen ONLY-Namen. Für letzteren wird keine eindeutige sondern eine möglichst *mehrdeutige* Identifizierung angestrebt. Dazu wertet man die potenziell mehrelementige Ergebnisliste von Algorithmus $\mathcal{I}$ aus (Kapitel 6). Darüberhinaus stellt sich die Frage in welchem Namensraum der partielle ONLY-Namen identifiziert werden soll:

```
IMPORT Real ONLY real
IMPORT Seq ONLY seq[real]    - illegal
```

Zur Identifizierung von `seq[real]` ist der generische Namensraum `Seq` nicht geeignet. Nur im umfassenden Namensraum, mit dem eine *Importinstanz* aufgelöst wird, ist `real` bekannt:

```
IMPORT Real ONLY real
IMPORT Seq[real] ONLY seq[real]    - legal
```

Im instanziierten Namensraum `Seq[real]` kann `seq[real]` identifiziert werden. Ohne den aktuellen Parameter `real` wäre der Namensraum `Seq[real]` nicht abgeschlossen. In OPAL wird `real` trotzdem nicht *reexportiert*:

```
IMPORT Real ONLY real
IMPORT Seq[real] ONLY real    - Reexport?
```

Der doppelte Import von `real` ist zwar nie nötig, aber muss deswegen nicht illegal sein. Offenbar verhindert die in `Seq` unbekannte Herkunft `Real` den Reexport oder der *aktuelle* Parameter `real` wird ebenso wenig exportiert wie ein formaler.

Mehrdeutigkeiten für *Importinstanzen* wurden in Abschnitt 7.1.2 diskutiert und verworfen. Die illegale Instanziierung des ONLY-Namen wird technisch verhindert, indem man *generische* Namen *speziell instanziiert* (Abschnitt 7.2.3). Dadurch kann z.B. das polymorphe `n`-Tupel nicht mit einem mehrdeutigen Typ `t` instanziiert werden und zu 2^n Lösungen führen.

Durch die Eindeutigkeit der Importinstanz kann sich eine Instanzannotation am ONLY-Namen nur für *reexportierte* Namen unterscheiden. Wie zuvor für die Herkunft kann man dann besser auf Instanzannotationen verzichten.

7.4.3 Reexport generischer Namen

Der instanziierte Import von `Seq`$[\alpha]$ innerhalb von `Set` bewirkt, dass der instanziierte Namensraum `Set`$[$`nat`$, <]$ nur instanziierte Namen enthält und insbesondere die Struktur `Seq`[`nat`] reexportiert wird. Wie sähe aber der Reexport aus, wenn `Seq` innerhalb von `Set` generisch importiert worden wäre?

In OPAL wurde dieses Problem lange durch die Trennung einer Struktur in Schnittstelle und Implementierungsteil sowie den ausschließlich *instanziierten* Importen für Schnittstellen vermieden. Die explizite Auflistung aller benötigten Instanzen erwies sich aber als lästig und deswegen wurden *generische* Importe für Schnittstellen erlaubt. Dieses hat sich in der Praxis relativ gut bewährt. Für den Reexport wurden folgende drei Möglichkeiten erwogen:

1. Es werden nur die konkreten Instanzen von `Seq` reexportiert, die innerhalb der Struktur `Set` *verwendet* werden.
2. Die Namen aus der Struktur `Seq` können transitiv nur generisch aus `Set` importiert werden.
3. Die generischen Namen aus `Seq` können transitiv beliebig instanziiert importiert werden.

Die Beschränkung auf konkret verwendete Instanzen ist äußerst unintuitiv, da erstens diese Instanzen an Applikationsstellen nicht so unmittelbar klar sind und zweitens eigentlich alle potenziellen Instanzen von `Seq` im Namensraum von `Set` liegen und deswegen auch reexportiert werden müssten. Schließlich werden in OPAL nicht benutzte instanziierte Importe ebenfalls reexportiert. Konsequenter würde man als Reexport den *minimal abgeschlossenen* Namensraum festlegen, der genau die gegebenen Typ- und Funktionsdeklarationen umfasst. Sobald eine Struktur dann instanziiert wird, sind alle reexportierten Namen *monomorph*!

Die zweite Alternative entspricht der OPAL-Implementierung und dem Konzept mit der generischen Instanz (Abschnitt 7.2.3). Aus der *instanziierten* Struktur `Set`$[\texttt{nat}, <]$ kann man die Sequenzen `seq` dann allerdings auch nur *generisch* transitiv importieren. Dadurch darf der generische Name zwar *instanziiert* in Annotationen vorkommen, wird aber nicht instanziiert sondern nur generisch reexportiert:

IMPORT `Set`$[\texttt{nat}, <]$ ONLY `seq asSeq`: `set`$[\texttt{nat}, <] \to$ `seq`$[\texttt{nat}]$

Weitere instanziierte Importe von `seq` wären nutzlos und damit instanziierte Importe insgesamt relativiert.

Die dritte Alternative, generische Namen beliebig instanziiert transitiv zu importieren, ist ebenso problematisch wie direkt aus der generischen Struktur `Seq` potenziell mehrdeutige Instanzen zu importieren (Abschnitt 7.4.2).

7.4.4 Teilinstanziierte Namen

Mit Hilfe von Reexporten können teilinstanziierte Namen importiert werden:

```
STRUCTURE S[α]
  IMPORT Pair[α, nat] COMPLETELY
```

```
STRUCTURE T
  IMPORT S COMPLETELY
```

Durch den generischen Import der Struktur S in T werden transitiv die Paare Pair teilinstanziiert importiert. Teilinstanziierte Namen im Namensraum haben den Nachteil, dass sie für bestimmte Instanzen überlappen können und deswegen zusätzlicher Aufwand bei der Namensidentifikation nötig wird (Abschnitt 6.2.2).

Legt man dennoch auf teilinstanziierte Namen Wert, dann sind Reexporte auf Grund ihrer Indirektion eher ein ungeeignetes Sprachmittel. Besser wären direkte Importe Pair[_, nat] mit Platzhaltern (vgl. Abschnitte 7.2.3, 7.1.2 und 8.4.3)

7.4.5 Namensraumexplosion

Der formale Parameter α der Struktur Set$[\alpha, <]$ bewirkt eine Abhängigkeit der reexportierten Struktur Seq$[\alpha]$. Durch mehrere Instanzen und längere transitive Importketten kann man daher exponentiell viele Namen erzeugen. Im Folgenden wurden Typdeklarationen für formale Parameter α und β weggelassen:

```
STRUCTURE S[α]
  TYPE a
  TYPE b

STRUCTURE S1[α, β]
  IMPORT S[α] ONLY a b
  IMPORT S[β] ONLY a b

STRUCTURE Si+1[α, β]
  IMPORT Si[α, β] ONLY a b
  IMPORT Si[a[α], b[β]] ONLY a b
  IMPORT Si[a[β], b[α]] ONLY a b
```

Die Struktur $\mathtt{S}_1$ enthält die 4 Typnamen a$[\alpha]$, a$[\beta]$, b$[\alpha]$, b$[\beta]$. In der Struktur $\mathtt{S}_2$ kommen zu den obigen vier noch die acht Namen a[a$[\alpha]$], a[b$[\beta]$], b[a$[\alpha]$], b[b$[\beta]$] und a[a$[\beta]$], a[b$[\alpha]$], b[a$[\beta]$], b[b$[\alpha]$] hinzu. Insgesamt umfasst $\mathtt{S}_{i+1}$ immer die gleichen Namen aus $\mathtt{S}_i$. Alle 12 Namen der Struktur $\mathtt{S}_2$ werden in $\mathtt{S}_3$ importiert; darüberhinaus werden diese 12 Namen noch zweimal erweitert

instanziiert importiert, wodurch zusätzlich $2 * 8 = 16$ *längere* (und verschiedene) Namen hinzukommen. S_3 umfasst also $12 + 16 = 28$ Namen. Für S_4 ergibt sich die Anzahl $28 + 2 * 16 = 60$ und allgemein lautet für S_i die Formel $2^{i+2} - 4$. (Die Instanziierungen in S_i sind immer eindeutig.)

Nicht nur der Namensraum wächst exponentiell mit der Anzahl der Strukturen sondern auch die *vielfach* importierten Namen, die nur einmal im Namensraum einzutragen sind. In der Praxis sind derartig gigantische Namensräume irrelevant aber allein die Möglichkeit ihrer Konstruktion ist eher schädlich als nützlich.

Nachdem die Mischung von generischen und instanziierten transitiven Importen problematisch war, sprechen hauptsächlich die Erfahrungen aus OPAL, also der höhere Schreibaufwand, gegen ausschließlich instanziierte Importe.

7.4.6 Abhängige Importinstanzen

Die Frage (Abschnitt 7.4.2) in welchem Namensraum ein partieller ONLY-Name zu identifizieren ist, soll nun noch für gewisse zyklische Importe untersucht werden:

```
IMPORT Seq[nat] ONLY nat ...
```

Der Typ `nat` wird aus `Seq` reexportiert und dann für die Instanziierung `Seq[nat]` benutzt. Ein Zyklus könnte auch mehrere Deklarationen umfassen, z.B. wenn die parametrisierten Strukturen `C` und `D` jeweils (die Typen) `b'B` bzw. `a'A` reexportieren:

```
IMPORT C[a] ONLY b
IMPORT D[b] ONLY a
```

Noch schwieriger wird es, wenn transitive Importe nur von einem Teil der Importinstanz abhängen. Im Folgenden stehen `a` und `b` für Typparameter sowie `c` und `d` für parametrisierte Typen:

```
STRUCTURE E[a, b]
  IMPORT C[a] ONLY c
  IMPORT D[b] ONLY d

STRUCTURE F[a, b]
  IMPORT E[c, b] ONLY d
  IMPORT E[a, d] ONLY c
```

Die obige Struktur `E[a, b]` reexportiert die Namen `c[a]` und `d[b]`. Aus `E[c, b]` kann also (unabhängig von `c`) `d[b]` und aus `E[a, d]` (unabhängig von `d`) `c[a]` transitiv importiert werden. Mit diesen Importen sind dann die Instanzen eindeutig zu `E[c[a], b]` und `E[a, d[b]]` auflösbar.

Eine inkrementelle Analyse (vgl. Kapitel 8) kann die formale Parameterliste in *unabhängig* identifizierbare Teile partitionieren und davon abhängige Reexporte vorab zum Namensraum hinzunehmen.

Die beiden Typparameter der obigen Struktur `E` sind *unabhängig*. Die Parameter der Struktur `Set` hingegen sind *abhängig* voneinander und müssen zusammen aufgelöst werden:

```
IMPORT Nat ONLY nat
IMPORT Set[nat, <] ONLY seq
```

Ohne Berücksichtigung der <-Relation darf der partielle Name `nat` nicht zu `nat'Nat` aufgelöst werden und dann transitiv aus `Set` der Name `seq[nat'Nat]` importiert werden. Möglicherweise ist die <-Relation *überlagert*, dann könnte sich die eindeutige Instanz `Set[nat'BigNat, <]` ergeben (Kapitel 5), wodurch tatsächlich `seq[nat'BigNat]` reexportiert würde.

Ein komplizierter Zyklus ergibt sich, wenn der Typ eines *abhängigen* Parameter reexportiert wird und sich erst daraus die Gesamtinstanz ergibt:

```
IMPORT Nat ONLY nat
IMPORT Set[nat, <] ONLY rel
FUN < : rel[nat]
```

Ohne die Kenntnis der Gesamtinstanz `[nat, <]` ist der Reexport `rel[nat]` unklar, und ohne `rel[nat]` fehlt auch die für die Instanz nötige <-Relation. Betrachtet man wieder die Überlagerung des Typs `nat`, dann kann sich die Auflösung der Instanz aus einer Annotation in der *entfernten* Funktionsdeklaration ergeben:

```
IMPORT Nat ONLY nat
IMPORT BigNat ONLY nat
IMPORT Set[nat, <] ONLY rel
FUN < : rel[nat'BigNat]
```

Die Analyse derartiger Zyklen wird in OPAL durch die Verwendung von Unbekannten für die nicht direkt auflösbare Instanz von `Set` versucht (Abschnitt 7.1.1). Die fehlerhafte *unabhängige* Betrachtung von `nat` in der Instanz `Set[nat, <]` führt aber zu einer Mehrdeutigkeit (Kapitel 5).

Im Sinne der nachfolgenden Namensraumanalyse und der lokalen Auflösbarkeit (Abschnitt 7.1.1) ist es angemessen und pragmatisch derartige Zyklen als Fehler zurückzuweisen.

Als Hauptvorteil für transitive Importe verbleibt, vollständig generisch große Namensräume zu bilden. In Abschnitt 8.4 werden dazu alternative Modulkonzepte beleuchtet.

Kapitel 8

Namensraumanalyse

Mit dem Werkzeug der Namensidentifikation $\mathcal{I}$ (Kapitel 6), bei der von einem festen Namensraum ausgegangen wird, ist nun die eigentliche *Konstruktion* der Namensräume zu untersuchen.

Ein Namensraum wird durch *Deklarationen* der Form δ ::== `ide` : `name` etabliert. Durch eine *Funktionsdeklaration* FUN `ide` : `name` wird ein neuer vollständiger Name mit dem Bezeichner `ide` in Abhängigkeit von dem partiellen Namen `name` *deklariert*. Im partiellen Namen `name` wiederum werden andere, schon bekannte Namen *appliziert*.

Typdeklarationen der Form TYPE `ide` sind völlig unabhängig von anderen Deklarationen. Freie Datentypvereinbarungen, wie z.B. für die Sequenzen `seq`, werden zu ihrer *induzierten Signatur* expandiert. Ebenso sind *Importe* (siehe Kapitel 7) als Deklarationen der Form IMPORT `name` ONLY `ide` zu betrachten, wobei durch Überlagerungen i.A. mehrere Namen mit dem Bezeichner `ide` importiert werden.

Insgesamt ergibt sich der Namensraum genau aus den mit TYPE, FUN oder IMPORT eingeleiteten *Deklarationen*. Dieser Namensraum selbst (bzw. nur ein Teilraum davon) ist die Basis für die Namensidentifikation der partiellen Namen in *Applikationen*. Die für die Namensraumanalyse relevanten *Applikationsstellen* sind dabei

- Typen in Funktionsdeklarationen und
- Instanzen von Importen.

Lineare Sichtbarkeit

Die elementarste Namensraumkonstruktion ist die sequenzielle Abarbeitung des Quelltextes und eine schrittweise Vergrößerung des Namensraums. Alle Namensidentifikationen hängen von der textuellen Position ab, wodurch eine gewisse Notationsreihenfolge abhängiger Deklarationen erzwungen wird. Diese *lineare Sichtbarkeit* ist für fast alle algebraischen Spezifikationssprachen verbreitet. OPAL bildet eine rühmliche Ausnahme!

Für einen Quelltext mit $\mathtt{n}$ Deklarationen $\delta_1 \ldots \delta_\mathtt{n}$ ergibt sich eine korrespondierende Folge von Namensräumen $\mathtt{N_0} \ldots \mathtt{N_n}$. Der initiale Namensraum ist dabei im einfachsten Fall leer; der nächste Namensraum $\mathtt{N_{i+1}}$ ergibt sich aus dem vorherigen Namensraum $\mathtt{N_i}$ und der in diesem Kontext (mit $\mathcal{I}$) zu identifizierenden Deklaration $\delta_{\mathtt{i}+1}$:

$$\mathtt{N_0} = \{\}$$
$$\mathtt{N_{i+1}} = \mathtt{N_i} + \mathtt{I}(\mathtt{N_i}, \delta_{\mathtt{i}+1})$$

Die Identifikation einer Deklaration $\delta = \mathtt{ide} : \mathtt{name}$ erfordert die *Namensidentifikation* von `name` und die Konstruktion eines *neuen* Namens mit dem Bezeichner `ide` und mit dem *Typ* oder der Importinstanz $\mathtt{I}(\mathtt{N}, \mathtt{name})$ (siehe Abschnitt 6.2).

Die schon für die imperative Sprache PASCAL verwendete sequenzielle (*One-Pass*) Analyse hat methodische Nachteile, die relativ leicht vermieden werden können. In den jüngeren imperativen, objektorientierten und funktionalen Sprachen mit einem Modulkonzept z.B. JAVA und HASKELL ist der Namensraum für ein Modul einheitlich. Der Gültigkeitsbereich der importierten und deklarierten Namen ist immer das gesamte Modul.

Die unschöne FORWARD-Deklaration aus PASCAL kann z.B. leicht durch einen zusätzlichen Durchlauf (*Pass*) vorab, bei dem alle Deklarationen aufgesammelt werden, oder nachträglich durch *Back-Patching* vermieden werden. Die Abhängigkeiten in den Signaturen von algebraischen Sprachen können allerdings tiefer geschachtelt sein, so dass mehrere Durchläufe erforderlich sind.

Im folgenden Abschnitt 8.1 wird eine *datengetriebene Namensraumanalyse* angegeben, die unabhängig von der textuellen Reihenfolge der Deklarationen ist und nur den hierarchischen Aufbau der Namen (aus Abschnitt 5.3) ausnutzt. In Abschnitt 8.2 wird die in OPAL vorbildlich *entworfene* – aber nur unvollständig implementierte – Deklaration der formalen *Parameter* erläutert und mit denen anderer Sprachen verglichen.

Die datengetriebene Namensraumanalyse erweist sich für die Konstruktion der abgeschlossenen Parametersignatur als besonders geeignet. Inspiriert von LPG werden als Neuerung *implizite* formale Parameter vorgeschlagen, womit auf elegante Weise die aus OBJ bekannten *Views* simuliert werden können (Abschnitt 8.3). Abschließend werden *zyklische* Strukturen mit strukturübergreifender polymorpher Rekursion und andere Modularisierungsaspekte betrachtet (Abschnitt 8.4).

8.1 Reihenfolgeunabhängigkeit

Als Motivation für die nachfolgende Namensraumanalyse betrachten wir zunächst die Abhängigkeitsanalyse mehrerer Strukturen bezüglich der *azyklischen* Importhierarchie (Abschnitt 7.4).

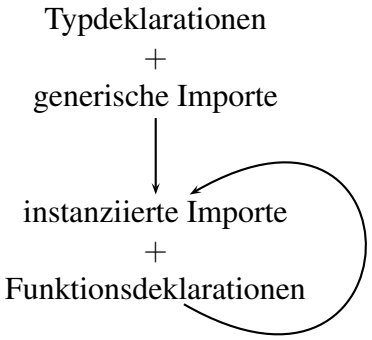
Liegen die Strukturen als getrennte Quelltextdateien vor, dann ist allein auf Grund der Dateinamen keine Analysereihenfolge bekannt. Man würde also alle Dateien durchsuchen und im ersten Schritt nur solche Strukturen `D` akzeptieren, die keine anderen Strukturen importieren. In weiteren Schritten würden dann solche Strukturen (`B` und `C`) analysiert, die nur schon bekannte Strukturen importieren. Diese fortzusetzende Vorgehensweise ist *datengetrieben*, da die bekannten Daten benutzt werden, um noch unbekannte Daten zu analysieren.

Eine alternative nahezu äquivalente Vorgehensweise ist *zielorientiert*. Ausgehend von einer (Toplevel-) Struktur `A`, die zu analysieren ist, würde man zunächst rekursiv die *importierten* Strukturen analysieren, die ihrerseits Rekursionen anstoßen. Dabei kann eine Struktur `D` mehrfach von verschiedenen Strukturen (`B` und `C`) importiert werden. Mit etwas Verwaltungsaufwand kann eine doppelte Analyse (von `D`) vermieden werden; außerdem sollte (statt einer Endlosrekursion) eine illegale zyklische Abhängigkeit erkannt werden können. Bei der datengetriebenen Abhängigkeitsanalyse werden zyklische Importe daran erkannt, dass die betroffenen Strukturen schlicht nicht analysierbar sind, da mindestens eine importierte Struktur noch nicht bekannt ist.

Die folgende Namensraumanalyse, die *unabhängig* von der textuellen Reihenfolge der Deklarationen ist und *Literate Programming* [Knu84] besser unterstützt, basiert auf dem induktiven Namensaufbau. Beginnend mit dem leeren Namensraum ergibt sich der nächste Namensraum $\mathtt{N}_{\mathtt{i}+1}$ aus allen Deklarationen, die mit Hilfe des `i`-ten Namensraums identifizierbar sind. Die folgende Notation $\mathtt{I}?(\mathtt{N}, \delta)$ steht für die eindeutige Identifizierbarkeit der Deklaration δ und $\mathtt{I}(\mathtt{N}, \delta)$ bezeichnet dann den zugehörigen vollständigen Namen:

$$\mathtt{N}_0 = \{\}$$
$$\mathtt{N}_{i+1} = \{\mathtt{I}(\mathtt{N}_i, \delta_j) \mid 1 \leq j \leq n \wedge \mathtt{I}?(\mathtt{N}_i, \delta_j)\}$$

Den Namensraum $\mathtt{N}_1$ bilden so die *generischen* und unparametrisierten *Importe* sowie die *Typdeklarationen*. Für solche Deklarationen müssen keine partiellen Namen identifiziert werden. Danach können einige Typen von Funktionsdeklarationen oder Instanzen von Importen identifiziert werden und die zugehörigen Namen zum Namensraum hinzugefügt werden. Mit diesem größeren Namensraum $\mathtt{N}_2$ können dann weitere Funktionalitäten oder Importinstanzen identifiziert werden:



Die iterative Berechnung der Namensraumfolge $\mathtt{N}_i$ muss so lange fortgesetzt werden, bis sie nicht mehr *streng* monoton wächst. Dieses ist nach spätestens n Iterationen der Fall, da in jeder Iteration mindestens eine weitere Deklaration δ_j zusätzlich identifiziert werden muss. Ändert sich der Namensraum nicht mehr, dann sind entweder alle Deklarationen aufgelöst oder unauflösbare (fehlerhafte) Deklarationen verblieben. In Abschnitt 8.1.2 wird ein korrektes Beispiel angegeben. Abschnitt 8.1.3 erläutert die Nachteile einer – alternativen und konkret für OPAL implementierten – *zielorientierten Namensraumkonstruktion* am Beispiel *wiederholter Deklarationen*.

Mehrdeutige Deklarationen sind ebenfalls Fehler, die direkt während der Identifizierungen erkannt werden können. Das Prädikat $\mathtt{I}?(\mathtt{N}_i, \delta_j)$ ist für eine mehrdeutige Deklaration δ_j nicht mehr erfüllt; d.h. eine Mehrdeutigkeit kann man auch daran erkennen, wenn ein Name $n \in \mathtt{N}_i$ nicht mehr in $\mathtt{N}_{i+1}$ vorkommt, also die Namensraumfolge bezüglich der Teilmengenrelation nicht mehr monoton wächst. Nach dem Entdecken einer Mehrdeutigkeit muss die Iteration abgebrochen werden bzw. die mehrdeutige Deklaration ignoriert (entfernt) werden.

8.1.1 Abschließender Mehrdeutigkeitstest

Eine kleine Variation bei der Konstruktion des Namensraums $\mathtt{N_{i+1}}$ ist die Berücksichtigung nur solcher Deklarationen, die noch nicht aufgelöst wurden. Deklarationen, die zuvor schon eindeutig aufgelöst wurden, bleiben eindeutig und werden erst ganz zum Schluss auf Mehrdeutigkeiten untersucht:

$$\mathtt{N_0} = \{\}$$
$$\mathtt{N_1} = \{\mathtt{I}(\{\}, \delta_\mathtt{j}) \mid \mathtt{1} \leq \mathtt{j} \leq \mathtt{n} \ \wedge \mathtt{I?}(\{\}, \delta_\mathtt{j})\}$$
$$\mathtt{N_{i+1}} = \mathtt{N_i} \cup \ \{\mathtt{I}(\mathtt{N_i}, \delta_\mathtt{j}) \mid \mathtt{1} \leq \mathtt{j} \leq \mathtt{n} \ \wedge \ \mathtt{I?}(\mathtt{N_i}, \delta_\mathtt{j}) \ \wedge \mathtt{I}(\mathtt{N_{i-1}}, \delta_\mathtt{j}) \notin \mathtt{N_i}\}$$

Bei dieser Variation wird nach dem Import von `seq[nat]` die partielle Instanz `Seq[seq]` (Abschnitt 7.1) als `Seq[seq[nat]]` aufgelöst, unabhängig davon, ob man `Seq[seq]` nachträglich als mehrdeutig zurückweist. Verzichtet man auf einen abschließenden Mehrdeutigkeitstest, dann wird zu Gunsten von hierarchisch untergeordneten Namen aufgelöst, was nicht unbedingt der Intuition entspricht:

```
IMPORT Nat ONLY nat
IMPORT Seq[nat] ONLY seq
IMPORT Seq[seq] ONLY seq
FUN f : seq
```

Im Beispiel würde die Instanz [`seq`] ebenso wie der Typ zur Funktionsdeklaration `f : seq` (scheinbar willkürlich) als `seq[nat]` aufgelöst. Die Zurückweisung von `seq` im Typ für `f` als mehrdeutig einerseits, aber eine eindeutige Akzeptanz von `seq` in der Instanz `Seq[seq]` andererseits, wäre methodisch wie implementierungstechnisch problematisch.

Die Reihenfolgeunabhängigkeit soll einen einheitlichen Namensraum für die gesamte Struktur garantieren; dieser Einheitlichkeit widerspräche eine lokale Beschränkung des Namensraums für die Analyse des Imports `Seq[seq]`, bei dem der damit importierte Name `seq` ausgeschlossen würde.

Der nach `k` Iterationen gewonnene Namensraum $\mathtt{N_k}$ mit $\mathtt{N_k} = \mathtt{N_{k-1}}$ sollte also auch diejenigen Deklarationen immer noch eindeutig identifizieren, die zuvor nur in einem kleineren Namensraum untersucht wurden:

$$\forall \mathtt{j}.\mathtt{I?}(\mathtt{N_k}, \delta_\mathtt{j})$$

Sind im Namensraum $\mathtt{N_k}$ alle Deklarationen eindeutig auflösbar, dann handelt es sich – auf Grund der *Monotonie* der Identifizierung aus Abschnitt 6.4 –

um genau die Auflösungen, die schon während der Iteration mit *kleineren* Namensräumen gefunden wurden und zur Konstruktion von $\mathtt{N_k}$ beigetragen haben. (Ein in einem kleineren Namensraum identifizierter Name kann in einem größeren Namensraum höchstens mehrdeutig geworden sein.) Für den finalen Namensraum gilt also:

$$\mathtt{N_k} = \{\mathtt{I}(\mathtt{N_k}, \delta_\mathtt{j}) \mid \mathtt{1} \leq \mathtt{j} \leq \mathtt{n}\}$$

Die inkrementelle Berechnung des Namensraums korrespondiert zum streng hierarchischen Aufbau der Namen und schließt zyklische Namen aus. Der abschließende Mehrdeutigkeitstest garantiert a-posteriori die Eindeutigkeit aller Identifizierungen!

Eine Fehlentscheidung bei der Identifikation eines Namens $\mathtt{n_j}$ in einem kleineren Namensraum wäre nur möglich, wenn eine noch nicht bekannte Mehrdeutigkeit vorliegt. Diese Mehrdeutigkeit wird aber entweder unabhängig von $\mathtt{n_j}$ entdeckt oder (wie für den zyklischen Import von `seq`) durch $\mathtt{n_j}$ selbst erzeugt. In beiden Fällen wird der Quelltext zurückgewiesen.

Ohne Monotonie der Namensidentifikation, etwa wenn deklarierte Namen importierte Namen (wie in PVS) verschatten, kann – abgesehen von der Verschattungsproblematik für die Namensidentifikation (vgl. 3.4) – aus der Eindeutigkeit *am Ende* nicht auf dieselbe Identifizierung *während* der Namenskonstruktion geschlossen werden.

8.1.2 Eindeutiges Beispiel

Im folgenden Beispiel `SetOfSet` muss zunächst der Elementtyp `elem` in Zeile 4 bekannt sein, um die zugehörige <-Ordnung in der letzten Zeile 6 auflösen zu können. Der Elementtyp und die Ordnung werden weiter in Zeile 2 als Instanz der Mengenstruktur `Set` *appliziert* und dadurch einfache Mengen $\mathtt{set}[\mathtt{elem}, <]$ importiert. In Zeile 5 wird für die importierten einfachen Mengen eine Ordnung deklariert, die in Zeile 3 eine weitere Instanziierung der Mengenstruktur erlaubt, wodurch Mengen von Mengen $\mathtt{set}[\mathtt{set}[\mathtt{elem}, <], <]$ importiert werden.

```
STRUCTURE SetOfSet                                              1

  IMPORT Set[elem, <] ONLY set : TYPE      - Mengen             2
  IMPORT Set[set[elem, <], <] ONLY set     - geschachtelt       3

  TYPE elem                                - unspezifiziert     4

  FUN < : set[elem, <] × set[elem, <] → bool   - Ordnungen      5
  FUN < : elem × elem → bool                                    6
```

Die Zeilen werden also gemäß dem Aufbau der Namen in der Reihenfolge 4, 6, 2, 5, 3 analysiert. *Dieselbe* Analyse ergibt sich für *parametrisierte Strukturen*. Man kann nachträglich vom Typ `elem` und von der zugehörigen Ordnungsfunktion *abstrahieren*, indem man diese als formale Parameter auflistet. Die Bezeichnung `elem` kann beibehalten werden:

```
STRUCTURE SetOfSet[elem, <]
            -weiter wie oben
```

Interessant ist noch die Überlagerung für die <-Relation über dem Elementtyp `elem` und den Mengen `set`[`elem`, <]. Auf Grund der *Abgeschlossenheit* der Parameter kommt die zu exportierende <-Funktion nicht als Parameter in Frage. *Zwei* formale <-Parameter müssten durch Typannotationen unterschieden werden:

```
STRUCTURE SetOfSet[elem, < : rel[elem], <: rel[set]]
            -wie oben
```

Es werden nur noch Mengen von Mengen *reexportiert*. Für einen *aktuellen* Elementtyp `elem` wäre natürlich der *direkte* Import `Set`[`set`[`elem`, <], <] adäquater. Die Struktur `Set` stellt darüberhinaus eine Ordnung für (beliebig tief) geschachtelte Mengen zu Verfügung (Kapitel 5).

Da der Typ `set`[`elem`, <] im dritten formalen Parameter <: `rel`[`set`] ebenso wie `rel`[`elem`] *importiert* wird und nur von untergeordneten formalen Parametern abhängt, ist diese Parametrisierung zwar fragwürdig aber *legal* (Abschnitt 8.2).

8.1.3 Wiederholte Deklaration

Dieselbe zielorientierte Vorgehensweise zur Analyse mehrerer Strukturen ist auch für die Namensraumkonstruktion innerhalb einer Struktur denkbar aber

etwas anspruchsvoller. Muss für irgendeine Deklaration δ_j der partielle Name $\texttt{name}_j$ identifiziert werden, dann sind rekursiv solche Deklarationen zu untersuchen, deren Bezeichner *möglicherweise* in $\texttt{name}_j$ appliziert werden. Die überlagerten Bezeichner erschweren bei dieser Vorgehensweise die Verwaltung bzw. erfordern Fallunterscheidungen, die datengetrieben einfacher (und modular) durch die *Namensidentifikation* behandelt werden.

Die datengetriebene Namensraumanalyse ist außerdem besser geeignet, wiederholte Deklarationen zu behandeln. Methodisch soll durch die redundante Wiederholung einer Deklaration die Lesbarkeit und Konsistenz verbessert werden. (Eine inkonsistente Änderung von nur einer Deklaration würde durch eine fehlende Definition bemerkt werden.)

Auf Grund der *partiellen* Schreibweise von Namen müssen wiederholte Deklarationen syntaktisch nicht identisch sein. Zwei Deklarationen sind dann gleich, wenn sie für denselben *vollständigen* Namen stehen.

Für die unteren zwei Funktionsdeklarationen `f` ist aber nicht offensichtlich, ob es sich um Überlagerung oder um eine Wiederholung handelt:

```
IMPORT T ONLY t : TYPE     - unparametrisiert

IMPORT S ONLY s : TYPE     - wie ist die Struktur S parametrisiert?
IMPORT R[t] COMPLETELY     - wird ein Typ s importiert?

FUN f : s'S
FUN f : s[t]
```

Abhängig von den Strukturen `S` und `R`, in die man hinein schauen müsste, wird *ein* `f : s'S[t]` *wiederholt* deklariert oder es werden *zwei* Funktionen `f` über verschiedenen (überlagerten) Typen `s`, etwa `s'S[]` und `s'R[t]`, deklariert.

Da bei der inkrementellen Analyse eine der beiden Deklarationen zuerst identifiziert wird, ist leicht erkennbar, ob die andere Deklaration mit einem schon bekannten Namen zusammenfällt. Bei der zielorientierten top-down Analyse wäre der Verwaltungsaufwand für scheinbar überlagerte Deklarationen, die später zusammenfallen, größer.

Analog zu wiederholten Funktionsdeklarationen sind auch wiederholte Importe möglich. Ebenso wie die obigen Typen von `f` könnten jetzt Instanzen von `Seq` zusammenfallen:

```
IMPORT T ONLY t : TYPE
                ...          - weitere Importe
IMPORT Seq[s'S] ONLY seq
IMPORT Seq[s[t]] ONLY seq
```

Durch eine syntaktische Kennzeichnung von wiederholten Deklarationen wäre es möglich, diese direkt als *Applikationen* von Originaldeklarationen zu betrachten und insbesondere von ansonsten mehrdeutigen Deklarationen zu unterscheiden:

```
IMPORT Nat ONLY nat
IMPORT Seq[nat] ONLY seq
FUN f : seq                      - mehrdeutig oder wiederholt?
IMPORT Seq[seq[nat]] ONLY seq
FUN f : seq[seq[nat]]
```

Die erste (später mehrdeutige) Deklaration von `f` würde bei der inkrementellen Namensraumanalyse (zunächst) zum konkreten Namen `f : seq[nat]` führen. Wenn aber die mehrdeutige Deklaration eine partiell notierte Wiederholung der unteren Deklaration sein soll, dann darf nur `f : seq[seq[nat]]` zum Namensraum hinzugefügt und erst anschließend die (volle) Namensidentifikation der *Funktion* `f : seq` (statt nur vom Typ `seq`) vorgenommen werden.

Nachdem Mehrdeutigkeiten entdeckt wurden, kann man die zugehörigen Deklarationen natürlich für eine neue Analyse ignorieren und später testen, ob es sich um Wiederholungen handelt. Die Kosten einer doppelten Analyse stehen aber in keinem guten Verhältnis zu dem Nutzen, wiederholte Funktionsdeklarationen *partieller* angeben zu können.

Für die in Abschnitt 7.1.1 skizzierte *globale* Analyse mit *Unbekannten* im Namensraum erhöht sich der Aufwand. Durch zunächst disjunkte Unbekannte für Wiederholungen ist der initiale Namensraum zu groß. Dadurch entstehen Mehrdeutigkeiten, die später, wenn Unbekannte aufgelöst werden, zu korrigieren sind. Beim Verkleinern durch *Gleichsetzen* von Unbekannten können aber *zyklische* Namen entstehen, die *extra* entdeckt und entfernt werden müssen. Durch Entfernen doppelter Namen können wiederum weitere Namen mit Unbekannten zusammenfallen.

Unbekannte und Mehrdeutigkeiten sind nicht nur für *Importinstanzen* 7.1.2 problematisch, sondern determinieren die zu importierenden Namen nur unzureichend. Für einen partiellen ONLY-Namen wird man daher von maximal vielen, potenziell passenden Namen ausgehen müssen.

Wiederholte Deklarationen und partielle ONLY-Namen können eine globale Analyse beträchtlich erschweren. Die lokale Auflösbarkeit der Instanzen und die eindeutige Identifizierung für *typannotierte* ONLY-Bezeichner *direkter* Importe erhöht erheblich die Transparenz wiederholter Deklarationen für Mensch und Maschine.

8.2 Abgeschlossene Parametersignatur

Eine abgeschlossene *Parametersignatur* wird häufig *separat* festgelegt, so z.B. in LPG und OBJ durch *Properties* bzw. *Theorien*:

THEORY `TotalOrder`
  TYPE α
  FUN $<: \alpha \times \alpha \to$ `bool`
  LAW ...

STRUCTURE `Set`
  ASSUME `TotalOrder`$[\alpha, <]$

Die Parametersignatur `TotalOrder` ist *unparametrisiert*. Durch ASSUME werden *neue* formale Parameter *deklariert*. Die Eigenschaften der Parameter sind die aus `TotalOrder`. Die Instanznotation `TotalOrder`$[\alpha, <]$ (aus LPG) ermöglicht nur eine *Umbenennung* der Bezeichner. Die Theorie `TotalOrder` kann in anderen Strukturen *wiederverwendet* werden. Für einfache Typparameter existiert eine *triviale* Theorie.

In PVS wird die gesamte Parametersignatur direkt in eckigen Klammern notiert, was für den Regelfall einer einfachen Parametrisierung kürzer ist. In OPAL-Syntax sieht das so aus:

STRUCTURE `Set`$[\alpha:$ TYPE$, <: \alpha \times \alpha \to$ `bool`$]$

Nötige Importe können dabei *innerhalb der formalen Parameterliste* notiert werden. (Durch den *impliziten Import* der 42 Seiten großen PVS-Prelude ist der explizite Import in Parameterlisten selten nötig.) Die Reihenfolge der Parameter muss in PVS der *linearen Sichtbarkeit* gehorchen.

Die obige formale Parameterliste ist auch für OPAL korrekt, die partiellen Namen $\alpha:$ TYPE und $<:$ `rel`$[\alpha]$ in der Liste sind allerdings nicht die *Deklarationsstellen* der Parameter sondern *Applikationsstellen*. Parameter in OPAL

müssen explizit innerhalb der Struktur deklariert werden. (Für obigen Spezialfall wäre natürlich eine Ausnahmeregelung möglich.) Die Typannotationen in formalen Parameterlisten sind redundant und helfen höchstens, Parameter von überlagerten *exportierten* Namen zu unterscheiden. Darüberhinaus kann die Reihenfolge der partiell notierten Parameter *in der formalen Parameterliste* frei und unabhängig von einer Reihenfolge der Deklarationen *innerhalb* der Struktur gewählt werden (Beispiel 8.1.2). Die Reihenfolge *aktueller* Parameter in Instanzen (etwa `Set2`$[<, \texttt{nat}]$) ist damit festgelegt:

STRUCTURE `Set2`$[< : \alpha \times \alpha \to \texttt{bool}, \alpha :$ TYPE$]$
  TYPE α
  FUN $< : \alpha \times \alpha \to \texttt{bool}$

Die eigentliche Namensraumkonstruktion ist durch die *nachträgliche* Betrachtung der formalen Parameterliste im Vergleich zu LPG und PVS nicht mehr offensichtlich. In diesen beiden Sprachen wird die garantiert abgeschlossene Parametersignatur ganz leicht *vorher* bestimmt.

Selbstverständlich können (in OPAL) nicht beliebige Deklarationen formale Parameter werden. Folgende Forderungen müssen erfüllt sein:

1. Formale Parameter sind *undefiniert*.
2. Die Parametersignatur ist *abgeschlossen*.

Um von Teilen einer Struktur *abstrahieren* zu können, sollten formale Parameter *variabel* oder *unspezifiziert* sein. Durch Definitionsgleichungen und freie Datentypdeklarationen werden Namen festgelegt.

Der *Abschluss* gewährleistet, dass die Parametersignatur eine *untergeordnete* Basis der Gesamtstruktur ist. Alle Teilkomponenten von formalen Parametern gehören zur Parametersignatur und in Abhängigkeit davon werden die zu exportierenden Namen deklariert.

Durch die schrittweise Vergrößerung des Namensraums (Abschnitt 8.1) kann man in jeder Iteration versuchen, die Parameter *vorab* zu bestimmen. Die identifizierten Deklarationen sind dabei per Konstruktion abgeschlossen! (Die abgeschlossene Parametersignatur muss in offenen Namensräumen nicht *transitiv* abgeschlossen *sichtbar* sein. Garantiert sind nur die Toplevel-Typen formaler Parameter im Namensraum enthalten.)

Eine *nachträgliche* Prüfung des Abschlusses für die in der Parameterliste identifizierten Namen ist mühsam. Für Funktionsparameter mit Funktionen

im Typ ist die OPAL-Implementierung prompt unvollständig. Mehrdeutigkeiten bei der Parameteridentifikation werden von OPAL nicht bezüglich Abgeschlossenheit untersucht.

Zum Beispiel betrachte man die Überlagerung der <-Funktion aus der Struktur `Set` (Kapitel 5). Der Typ `set` sieht zwar syntaktisch ebenso aus, wie der formale Parameter α; er kann aber direkt als Parameter ausgeschlossen werden, weil er nicht in der Parameterliste vorkommt; die von `set` *abhängige* Funktion <: `rel`[`set`] kann deswegen erst recht kein formaler Parameter sein.

Die inkrementelle Namensraumanalyse unterstützt also ideal die Bestimmung der abgeschlossenen Parametersignatur.

Parameterannotation

Da formale Parameter vorab bekannt sind, lässt sich die folgende Notationsalternative für parametrisierte Namen innerhalb ihrer Herkunftsstrukturen realisieren:

FUN `#`: `seq`[α] $\rightarrow$ `nat` - *Notationsalternative*

Innerhalb der Struktur `Seq` ist `seq` *unparametrisiert*, kann also nicht instanziiert notiert werden. In Kenntnis des Parameters α ist es aber kein Problem, den *monomorphen* Namen `seq`[α] zu bilden. Konkret könnte man von einem *hypothetischen* Import `Seq`[α] ausgehen, nach dem `seq` mit oder ohne Instanz notiert werden darf.

Für die *unparametrisierten* formalen Parameter selbst sollte man keine Instanzannotation erlauben: eine *partielle* Notation könnte beliebig lang werden: $\alpha[\alpha[\alpha]]$. Eine Herkunftsannotation α'`Seq` kann dagegen Überlagerung mit einem *importierten* Typ α *an anderen Stellen* auflösen. Ein importierter Name α kann kein Parameter sein (nur ein Teil davon). Innerhalb der formalen Parameterliste sind daher *nur Typannotationen* sinnvoll.

Durch vorab bekannte formale Parameter können außerdem zyklische Strukturen unterstützt werden (Abschnitt 8.4.1). Für die nachfolgenden *impliziten* Parameter ist die Ermittlung der abgeschlossenen Parametersignatur ebenfalls leicht. Man sucht den kleinsten Namensraum, der alle aufgelisteten Parameter auflöst, und entnimmt *untergeordnete* Parameter.

8.3 Implizite Parameter

Für voneinander abhängige Parameter soll jetzt noch die durch LPG inspirierte Sprachvariante *impliziter* formaler Parameter diskutiert werden. Ein impliziter formaler Parameter muss zwar deklariert werden, kann aber in der Parameterliste fehlen, wenn er Teil eines *übergeordneten* Parameters ist. Die Struktur `Set` und zugehörige Instanzannotationen werden dadurch kürzer:

```
STRUCTURE Set[<]
          - weiter wie oben
```

Allein weil in der Deklaration des <-Parameters der deklarierte Typ α vorkommt und formale Parameter abgeschlossen sein müssen, wird der Typ α zum *impliziten* formalen Parameter. Bei einer Instanziierung `set[<'Nat]` wird entsprechend der implizite aktuelle Parameter dem Typ des aktuellen Funktionsparameters entnommen. Zwar geht dabei offensichtlich redundante Information verloren, diese kann man aber bei Bedarf durch Annotationen am übergeordneten Parameter kompensieren. Statt `Set[nat, <]` könnte man z.B. auch `Set[<: rel[nat]]` schreiben.

Die unannotierte Instanziierung für Mengen von Mengen `set[<[<]]` wäre allerdings mehrdeutig. Nur durch `set[<[<[]]` könnte die Mehrdeutigkeit zu Gunsten der *unparametrisierten* Funktion `<'Nat` aufgelöst werden. Geht man auch für `SeqOrd` von impliziter Parametrisierung aus, dann würde man die Namen am besten mit ihrer Herkunft annotieren:

```
set[<'SeqOrd[<'SeqOrd[<'Nat]]]
set[<'Set[<'Set[<'Nat]]]
```

Das *optionale* Weglassen untergeordneter Parameter *nur* bei Instanziierungen – also die gemischte Notation `set[<[nat, <]]` – würde die Überlagerungsauflösung allein an Hand der Anzahl der Parameter erschweren und die Wahrscheinlichkeit für Mehrdeutigkeiten erhöhen.

Am besten man überlässt es dem Benutzer, sich für eine feste Anzahl, Reihenfolge und den Grad der Redundanz durch die *formale* Parameterliste zu entscheiden! (Denselben Parameter mehrmals anzugeben, ist in OPAL verboten.) Theoretisch könnte man sogar auf völlig redundanzfreie Parameter bestehen.

Ein Problem der fehlenden Redundanz innerhalb von `Set` offenbart die Überlagerung der <-Funktion für den Typ `set`, der ebenso wie α impliziter Pa-

rameter sein könnte. Dabei hilft die Typannotation `Set`[<: `rel`[α]] in der formalen Parameterliste. Eine weitere Alternative wäre obige instanziierte Notation innerhalb von `Set`:

FUN <: `set`[<] × `set`[<] → `bool`

In diesem Fall wäre *offensichtlich* die Funktion <: `rel`[α] ein Teil der Funktion <: `rel`[`set`] und der Name `set`[<] kann kein Parameter sein, weil formale Parameter *unparametrisiert* sind.

Gegebenenfalls schließen Definitionsgleichungen Mehrdeutigkeiten mit Parametern aus (vgl. `SeqOrd` aus Abschnitt 1.3.1). Weiterhin ist auch die direkte Kennzeichnung formaler Parameter per Position oder Schlüsselwörter vorstellbar.

Notationell sparsam sind implizite Parameter besonders für tiefere Abhängigkeiten. Dazu betrachten wir die (zu `Foo` analoge) Struktur `SafeSet` aus Kapitel 5:

```
STRUCTURE SafeSet[proof]
  TYPE α
  FUN <: α × α → bool
  IMPORT TotalOrder[α, <] ONLY totalOrder
  FUN proof : totalOrder                    - Annahme
```

Die nachfolgende *Theorie* `TotalOrder` ist eine *Struktur* (oder SIGNATURE) und die *Eigenschaft* `totalOrder` ist ein *Typ* und der Beweis `proof` eine *Konstante*. Für die Theorie wird auf implizite Parameter verzichtet:

```
THEORY TotalOrder[α, <]
  TYPE α
  FUN <: α × α → bool
  LAW totalOrder          - Eigenschaft ist TYPE

STRUCTURE Nat
  TYPE nat = ...
  FUN <: nat × nat → bool
  IMPORT TotalOrder ONLY totalOrder
  FUN Ord : totalOrder[nat, <]          - Behauptung
```

Eine typkorrekte Instanz ist nun `SafeSet`[`Ord`]. Mit einer weiteren Deklaration `Ord` : `totalOrder`[`set`, <] in `SafeSet` kann man tiefer geschachtelte Mengen instanziieren: `SafeSet`[`Ord`[`Ord`[`Ord'Nat`]]].

Der *Name* `Ord` steht für einen Beweis und entspricht einem *View* aus OBJ. Der in OBJ umständliche Morphismus zwischen der formalen Theorie und den aktuellen Parametern aus `Nat` wird hier elegant durch Parameterlisten gebildet.

Der Beweis, dass die Definition von $<'$`Nat` tatsächlich die in der Theorie geforderte Eigenschaft `totalOrder` erfüllt, ist ein anderes Thema. Dem *Typ* `totalOrder` fehlt bisher eine *Definition*, die besagt, welche Formeln zu erfüllen sind; `totalOrder` korrespondiert insofern zu einem *Datentyp*. Ein konkreter Beweis korrespondiert zu einer Definitionsgleichung für `Ord`. Die Logik und Syntax eines Beweisers umfasst zumindest die konstruktiven Implementierungen (für `nat` und $<$) und ist nicht Inhalt dieser Arbeit.

Allein die Behauptung oder *Beweisdeklaration* `Ord` soll die statische Sicherheit vergrößern. Das entspricht der in [DGMP97] diskutierten Erweiterung OPAL 2 mit *Theorien*:

STRUCTURE `Set`$[\alpha, <]$
  ASSUME `TotalOrder`$[\alpha, <]$

Die *Annahme deklariert* die formalen Parameter mit ihren Eigenschaften vergleichbar zu LPG. (Für implizite Parameter `TotalOrder`$[<]$ und `Set`$[<]$ wäre diese Deklarationsform für eine Benennung von α ungünstig.)

Für die Korrektheit der Instanz `Set`$[$`nat`$, <]$ muss eine entsprechende *Behauptung* `TotalOrder`$[$`nat`$, <]$ *existieren*, die mit einer Instanzdeklaration aus HASKELL vergleichbar ist. In HASKELL wird für den Elementtyp eine Instanz zur Typklasse `Ord` benötigt, der dann die Implementierung der $<$-Funktion entnommen wird, um über Mengen zu operieren.

Die *Suche* nach einer Instanzdeklaration, bzw. einer geeignet zu instanziierenden Behauptung, kann durch potenzielle *Überlappungen* aber Entscheidbarkeitsprobleme aufwerfen [VS91].

Durch die Parametrisierung in `SafeSet` wird die benötigte Information, die Implementierung und der Beweis, explizit *benannt* und übergeben. Die impliziten Parameter unterstützen lediglich eine kürzere Notation langer Parameterlisten, die nun mit OBJ und HASKELL vergleichbar wird. Wie Datentypen in PVS könnte man auch die Theorie `TotalOrder` direkt wie ein Typ betrachten:

```
STRUCTURE Set[p]
  TYPE α
  FUN <: α × α → bool
  ASSUME p: TotalOrder[α, <]      - Parameter
  TYPE set
  FUN <: set × set → bool
  ASSERT p: TotalOrder[set, <]    - Nicht-Parameter!
```

Die Schlüsselwörter ASSUME und ASSERT sind als Synonyme für FUN zu betrachten, die sowohl die Unterscheidung von Theorien und echten Typen als auch von Parametern und Nicht-Parametern ermöglichen. Ob die mögliche Gleichbezeichnung, also Überlagerung[1] von Strukturen, Funktionen, Typen und/oder Beweisen sinnvoll ist, muss die Praxis zeigen.

8.4 Modulkonzepte

Die algebraischen Strukturen (in PVS Theorien) sind für die Modularisierung in einigen Fällen zu feinkörnig, da zumindest für unterschiedliche Parametrisierungen verschiedene Strukturen verwendet werden müssen. Geht man aber davon aus, dass ein Großteil der Strukturen unparametrisiert ist und darüberhinaus die meisten parametrisierten Strukturen durch eine Standardbibliothek zur Verfügung gestellt werden, dann sind die Strukturen fast mit dem einfachen Modulkonzept von HASKELL vergleichbar und zwar erst recht, wenn man den Import *überlagerter* Strukturen hinzunimmt (Kapitel 7). Eine zusätzliche Modularisierungsebene ist für viele kleine und mittlere Projekte also durchaus entbehrlich.

Ein Modulkonzept soll in erster Linie eine Gliederung in überschaubare – möglichst wiederverwendbare und getrennt übersetz- und beweisbare – Teile, eben Pakete oder Module, unterstützen, die sich eher am Problem (oder der Problemklasse) und weniger an Formalitäten, wie z.B. am Grad der Polymorphie, orientieren. Für OPAL 2 wurden dafür in [DGMP97] einfache *Pakete* vorgeschlagen, in denen mehrere Strukturen sogar wechselseitig voneinander abhängig sein können (Abschnitt 8.4.1). Anschließend wird ein hierarchisches Konzept diskutiert, das mit dem von SML vergleichbar ist (Abschnitt 8.4.2). Wie am parametrisierten Typsynonym `rel`$[\alpha]$ deutlich wurde, sind abkürzende Synonyme wichtige Strukturierungshilfen (Abschnitt 8.4.3). Zuletzt wird die Namensraumanalyse für getrennte Schnittstellen und Implementierungen, wie sie konkret für OPAL benötigt wird, skizziert (Abschnitt 8.4.4).

[1] Mit implizitem Parameter wäre `TotalOrder`$[<]$ innerhalb von `Set`[p] mehrdeutig.

8.4.1 Zyklische Strukturen

Die Motivation für die Zusammenfassung mehrerer Strukturen zu einem Paket war eine Abschwächung der streng azyklischen Importrelation von Strukturen. Nur noch ganze Pakete müssen eine azyklische Importhierarchie bilden, während sich die Strukturen innerhalb eines Pakets auch wechselseitig importieren können, bzw. alle Strukturen innerhalb eines Pakets gleichberechtigt bekannt sind. Zu beachten ist, dass trotz zyklischer Strukturen die Namen nach wie vor azyklisch bleiben.

Die Erweiterung der Signaturanalyse für parallel zu analysierende Strukturen ist durch die inkrementelle Namensraumanalyse (Abschnitt 8.1) und die frühzeitige Kenntnis der Parametersignatur (Abschnitt 8.2) leicht möglich. Zunächst werden die formalen Parameter irgendeiner Struktur ermittelt. Mit der Kenntnis von *allen* formalen Parametern mindestens einer Struktur werden dann einzelne Deklarationen dieser Struktur identifiziert und *direkt exportiert*.

Selbstverständlich kann ein formaler Funktionsparameter vom Import einer Struktur abhängen, dieses bedeutet aber nur, dass dieser importierte Name *zuerst* bekannt sein muss. Stammt dieser Name selbst aus einer *parametrisierten* Struktur, dann müssen dort die formalen Parameter zuerst analysiert werden. Ohne Kenntnis der formalen Parameter ist auch der *Export* unbekannt. Ein illegaler Zyklus liegt vor, wenn die formalen Parameter wechselseitig voneinander abhängen:

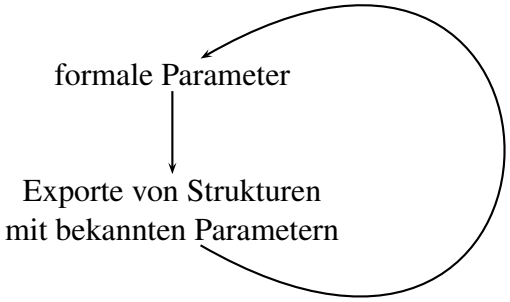
```
PACKAGE M                 - illegaler Zyklus

STRUCTURE A[f : b]        - benötigt b'B
  TYPE a
  FUN f : a

STRUCTURE B[f : a]        - benötigt a'A
  TYPE b
  FUN f : b
```

Der Typ `b` (resp. `a`) im formalen Parameter `f` von `A` (bzw. `B`) soll aus `B` (resp. `A`) stammen. Ein Teil der Parametersignatur `b` kann nicht von der exportierten Funktion `f : a` abhängen. Durch `b[f : a]` müssten der Typ `a` und `f : a` ebenfalls formale Parameter von `A` sein.

Ein *impliziter Parameterabschluss* (Abschnitt 8.3) hätte hier zur Folge, dass die Struktur `A` nichts exportieren, sondern nur Parameter enthalten würde.

Die Struktur `B` kann `a'A` nicht importieren, da `a'A` nicht exportiert wird, und deswegen kann der Abschluss zum Funktionsparameter `f` aus `B` nicht gebildet werden.

Für die simultane inkrementelle Signaturanalyse mehrerer Strukturen kann am Anfang also immer eine Struktur gefunden werden, deren formale Parameter alle vorab bekannt sind; andernfalls kann ein Zyklusfehler gemeldet werden. Eine derartige Struktur muss nun mindestens einen identifizierbaren Namen exportieren, der dann zur Auflösung von formalen Parametern anderer Strukturen beitragen kann. Der schematische Analyseablauf sieht graphisch wie folgt aus:



Die Behandlung unparametrisierter Strukturen ist natürlich nur ein einfacher Spezialfall und mehrere unparametrisierte Strukturen würden genau wie eine einzige große Struktur analysiert. Sobald Namen vollständig bekannt sind, tragen sie zur Identifizierung von hierarchisch übergeordneten Namen oder auch zu Mehrdeutigkeiten bei.

Polymorphe Rekursion

Zyklisch importierte Strukturen ermöglichen auch wechselseitig rekursive Definitionsgleichungen über Strukturgrenzen hinweg. Für zwei parametrisierte Funktionen `f'F` und `g'G` können rekursive Aufrufe der jeweils anderen Funktion beliebig instanziiert sein:

```
STRUCTURE F[α : TYPE]
  FUN f : ...
  DEF f(x) = g[α × α](x) ...

STRUCTURE G[α : TYPE]
  FUN g : ...
  DEF g(x) = f[α × α](...)
```

Die mit α parametrisierten Funktionen `f` und `g` werden im Beispiel rekursiv mit der Instanz $\alpha \times \alpha$ appliziert, d.h. es liegt *wechselseitige polymorphe* Rekursion vor. Obwohl die beiden Strukturen `G` und `F` hier gleich parametrisiert sind, kann diese polymorphe Rekursion *innerhalb einer* Struktur nicht notiert werden. Wenn aber zwei Strukturen eines Pakets wechselseitige polymorphe Rekursion erlauben, sollte erst recht die *direkte* polymorphe Rekursion möglich sein.

Analog zur Notationsalternative des Typkonstruktors `seq`[α] innerhalb von `Seq` (Abschnitt 8.2) kann man statt des hypothetischen Imports der Instanz `Seq`[α] auch den generischen Import `Seq` betrachten. Sobald also ein exportierter Name identifiziert wurde, wird er nicht *unparametrisiert* zum Namensraum hinzugefügt, sondern *generisch*. Die zu bindenden formalen Parameter sind alle bekannt. Eine *uninstanziierte* Notation `seq` innerhalb von `Seq` würde dadurch allerdings mehrdeutig. Dafür könnte man aber folgende `flatten`-Funktion für Sequenzen innerhalb der Struktur `Seq` definieren:

FUN `flatten`: `seq`[`seq`[α]] $\rightarrow$ `seq`[α]

Die Instanz `seq`[`seq`[α]] und ebenso die für eine Definition nötigen Funktionsinstanzen wären in der Struktur `Seq` illegal. Insofern ist weniger die *Rekursion* als die *Polymorphie* das Hauptentscheidungskriterium.

Insgesamt ist die Integration der polymorphen Rekursion intuitiv und im Gegensatz zu HASKELL *entscheidbar*, weil die Typsignaturen für alle Funktionen *vorgegeben* sind. Eine rekursive Aufrufstelle wird wie die Applikation einer *importierten* polymorphen Funktion analysiert.

Bei einem Verzicht auf Typsignaturen bliebe die Analyse für *monomorph* rekursive Funktionen entscheidbar. (Unproblematisch sind *nicht-rekursive* Funktionen.) Für monomorph rekursive Funktionen ohne Typsignaturen benötigt man Unbekannte, die mit Wiederholungen (Abschnitt 8.1.3) nicht kollidieren, da *undeklarierte* Funktionen nicht wiederholt deklariert werden können.

Die Gruppierung von gleichberechtigen Strukturen in Pakete legt eine Spracherweiterung um polymorphe Rekursion nahe. Der Preis sind mehr Instanzannotationen innerhalb parametrisierter Strukturen. Denkbar sind *Annotationen* für nicht-rekursive, rekursive bzw. polymorph-rekursive Namen. Die Kennzeichnung rekursiver Funktionen ist z.B. in PVS Pflicht. Für derartige Fragen müssen sich Sprachentwürfe in der Praxis bewähren.

8.4.2 Modulhierarchie

Für OPAL 2 [DGMP97] wurde eine einzige zusätzliche Modularisierungsebene durch Pakete als ausreichend empfunden. Darüberhinaus könnte man sich hierarchische Pakete und Unterpakete vorstellen, die sogar parametrisiert sein können:

```
PACKAGE Prelude

  STRUCTURE Nat                          - Struktur im Hauptpaket
  PACKAGE Seqs[α : TYPE]                 - parametrisiertes Unterpaket
    STRUCTURE Seq
    STRUCTURE SeqMap[β : TYPE]           - erweitert parametrisiert
    STRUCTURE SeqOrd[<: rel[α]]
```

Parametrisierte Module orientieren sich wieder eher an der Parametrisierung als am Inhalt und auf der obersten Ebene werden vermutlich meistens unparametrisierte Module verwendet. Dieser Nachteil gilt auch für das vergleichbare Funktorkonzept von SML, das zwar modular auch auf andere Kernsprachen angewendet werden kann [Ler00], aber nicht viel mit der Kernsprache zu tun haben muss. Insbesondere unterstützt SML dadurch die Polymorphie (vgl. Abschnitt 6.2) und Funktionen höherer Ordnung (vgl. Abschnitt 5.5) auf zwei Weisen: durch parametrisierte Strukturen und durch die Kernsprache ML.

Die hierarchischen Modulkonzepte kann man sich außerdem als eine Verteilung des Quelltextes in ein hierarchisches Dateisystem vorstellen. Wie in JAVA kann dann mit $*$-*Wildcards* der Import `java.lang.`$*$ notiert werden.

Flexibler wäre es, wenn man beliebige Strukturen und Untermodule zu Modulen zusammenfassen könnte. Module wären damit *nicht* notwendig *disjunkt*. Das ist auch ein Vorteil der Reexporte (Abschnitt 7.4). Das gesamte Modulsystem könnte dann als eine separate (und zentralverwaltete) Zuordnung von Strukturen (und Untermodulnamen) zu Modulnamen betrachtet werden. Strukturen könnten zwar direkt Module importieren, aber ihre eigene Zugehörigkeit zu Modulen würde nur indirekt über eine zentrale Modulzuordnungsdatei festgelegt, in der auf den *Ort* der Struktur verwiesen wird. Der Verweis könnte einfach ein absoluter oder relativer Pfadname des Dateisystems (oder des Internets) sein.

8.4.3 Synonyme

Ein interessanter Aspekt zur Strukturierung sind noch Synonyme, die in OPAL nicht zur Verfügung stehen. Außer Typsynonyme wie $\texttt{rel}[\alpha]$ wären Synonyme für ganze Strukturinstanzen sinnvoll:

STRUCTURE $\texttt{String} = \texttt{Seq}[\texttt{char}]$
  TYPE $\texttt{string} = \texttt{seq}[\texttt{char}]$ - *Typsynonym*

Statt z.B. die Konkatenationsfunktion $+\!\!+$ für eine Liste von Zeichen instanziiert $+\!\!+[\texttt{char}]$ zu notieren, kann man die Herkunftsannotation $+\!\!+{}'\texttt{String}$ ohne Instanz verwenden. Durch das Typsynonym kann auch der Typ entsprechend annotiert werden:

$+\!\!+ : \texttt{string} \times \texttt{string} \to \texttt{string}'\texttt{String}$

Eine gemischte Notation $\texttt{seq}\times\texttt{string}$ wird verhindert, wenn nur das Typsynonym `string` im Namensraum direkt sichtbar ist. Selbstverständlich sollten auch parametrisierte oder *teilinstanziierte* Synonyme möglich sein:

STRUCTURE $\texttt{NatMap}[\beta] = \texttt{Map}[\texttt{nat}, <' \texttt{Nat}, \beta]$

Nach einem Import von `NatMap` kann der Typ $\texttt{map}'\texttt{Map}$ auch $\texttt{map}'\texttt{NatMap}$ oder noch besser spezialisiert $\texttt{map}[\texttt{string}]$ notiert werden. Entsprechend spart man für Funktionen längliche Instanzlisten.

Der Import teilinstanziierter Namen (Abschnitt 7.1 und 7.4.4) bedurfte einer Zusatzbehandlung, um ungerechtfertigte Mehrdeutigkeitsfehler durch *überlappende* Namen zu vermeiden (Abschnitt 6.2.2). Für *überlappende* Synonyme sind aber Mehrdeutigkeitsfehler sogar *erwartungskonform* und eine Zusatzanalyse wäre entbehrlich!

Nützlich sind auch *curried* Parameter aus CASL [BM00] und Synonyme:

STRUCTURE $\texttt{Map}[\alpha, <][\beta]$
STRUCTURE $\texttt{NatMap} = \texttt{Map}[\texttt{nat}, <]$
STRUCTURE $\texttt{NatStrings} = \texttt{NatMap}[\texttt{string}]$

Durch Synonyme können wie für Substitutionen exponentiell große Terme repräsentiert werden:

STRUCTURE $\texttt{Twin}[\alpha] = \texttt{Pair}[\alpha, \alpha]$
  TYPE $\texttt{twin} = \texttt{pair}[\alpha, \alpha]$ - *Typsynonym*

Unabhängig vom konkreten Nutzen könnte ein Typ wie `twin`[`twin`[...]] kaum expandiert notiert werden. Implementierungstechnisch kann man aber Substitutionen erst bei Bedarf applizieren. Eine vorzeitige Expansion von Synonymen ist außerdem für das *Matchen* von Eigenschaften ungünstig, die expandiert nicht erkannt werden und damit verloren gehen. Idealerweise werden alle Substitutionen bis zur Unifikation durchgereicht und Terme als *DAG* verwaltet. Diese Behandlung passt hervorragend zu der fast-linearen Martelli-Montanari-Unifikation, die endlich mal konsequent in die Gesamtanalyse zu integrieren wäre!

8.4.4 Schnittstelle und Implementierung

Eng verbunden mit der Modularisierung ist die Kapselung und/oder das Verstecken von Information (*information hiding*). Für den nicht-freien Datentyp `set` (Kapitel 5) wird so die Implementierung z.B. durch sortierte Sequenzen abgetrennt. In HASKELL wird dazu der *Export* eines Moduls festgelegt. In JAVA können zu diesem Zweck die Entitäten, z.B. Methoden in Klassen und Paketen, als privat (*private*), geschützt (*protected*) und öffentlich (*public*) attributiert werden.

Die MODULA- und OPAL-ähnliche Zerlegung in eine Schnittstelle (*Interface*) und einen Implementierungsteil ist kaum noch verbreitet. Für OPAL 2 wird in [DGMP97] eine flexible *Implementierungsrelation* angestrebt, die nicht nur eine schrittweise Implementierung oder Verfeinerung erlaubt. Es ist vorgesehen, Instanzen einer Struktur `Seq`[`char`] *separat* und potenziell effizienter (aber *uniform*) zu implementieren. Obige Benennung der Instanzen durch *Synonyme* (wie `String`) wäre dafür sicher hilfreich. Darüberhinaus kann eine Struktur aus OPAL 2 auch mehrere Schnittstellen haben.

Es liegt nahe, beim Namensraum für eine Implementierung einfach vom Namensraum der zugehörigen Schnittstelle auszugehen. Besser ist es aber, den Namensraum der Implementierung nicht unbedingt mit den *transitiv* vorhandenen Namen der Schnittstelle zu belasten; das ist vergleichbar mit direkten Importen (Abschnitt 6.3). Nur was explizit benötigt wird, muss importiert werden! In Schnittstellen würde man dann hauptsächlich Typen für die Signaturen und in Implementierungen Funktionen (und keine Typen) für die Definitionsgleichungen importieren.

Kapitel 9

Ergebnisse

Mit der Namensidentifikation (aus Kapitel 6) und der Namensraumanalyse (aus Kapitel 8) wurde die Signaturanalyse für eine algebraische Sprache angegeben. Der Entwurf dieser Sprache ist *modern* in Bezug auf die zu den Algorithmen korrespondierende Notation von

- Instanziierungsmorphismen und
- Deklarationen in Namensräumen

Der Gültigkeitsbereich von Deklarationen ist immer die ganze Struktur und positionsunabhängig. Dennoch kann der Namensraum durch mehrere *Pässe* inkrementell analysiert werden.

Die Vorstellung von einer globaleren Namensraumanalyse, die als globale Suche nach Namensvervollständigungen spezifiziert wurde, wird hier präzisiert (Abschnitt 7.1.1). Sie betrifft nicht die unabhängige Reihenfolge von Deklarationen, sondern nur eine mächtigere, aber unnötige Überlagerungsauflösung. Erwartungsgemäß und konkret für Opal so implementiert ist aber *lokale Auflösbarkeit*.

9.1 Algorithmus $\mathcal{I}$

Kern dieser Arbeit ist der Algorithmus $\mathcal{I}$ zur Namensidentifikation, der den klassischen Hindley-Milner-Algorithmus $\mathcal{W}$ für funktionale Sprache um zwei Dimensionen verallgemeinert:

- Namensterme für Typen *und Funktionen*
- parametrische und Ad-hoc-Polymorphie

Beschränkt auf Typterme verbleibt ein Algorithmus $\mathcal{W}_o$ (Kapitel 4) als Verschmelzung von Algorithmus $\mathcal{W}$ und brute-force Überlagerungsauflösung. Algorithmus $\mathcal{W}_o$ erlaubt die Typanalyse polymorpher Funktionen mit Überlagerung, die der Uniformität durch Parametrisierung voll gerecht wird. Die Uniformität geht verloren für folgende zwei Betrachtungsweisen:

algebraisch: instanziierte Funktionen als Überlagerungen (Abschnitt 1.1.1)

funktional: Überlagerungen als *eine* Funktion mit einem, ggf. durch Antiunifikation berechneten, minimal *polymorphen* Typ (Abschnitt 4.3)

Den Dreh- und Angelpunkt algebraischer Sprachen bilden instanziierte Namen in einem Namensraum mit überlagerten und generischen Namen.

Der Algorithmus $\mathcal{I}$ arbeitet top-down und ist dadurch potenziell effizienter. Dem entsprechen die top-down Cormack-Variante zur Überlagerungsauflösung und die polymorphe Typinferenz mit Algorithmus $\mathcal{M}$ (Kapitel 2). Es ergeben sich Analogien für:

- Namen und Ausdrücke
- Instanzen und Typen
- Instanziierung und Funktionsapplikation

Die Analogie unterstützt Sprachentwurfsentscheidungen in Richtung Monomorphie (Abschnitte 4.4 und 7.1.2) und lokale Auflösbarkeit (Abschnitte 4.2, 4.3 und 7.1.1).

Namensterme

Eine entscheidende Grundlage bilden Namensterme (Kapitel 5) und ihre *Unifikation*. Pathologische Überlagerungen innerhalb einer Struktur können z.B. rechtzeitig mittels Unifikation ausgeschlossen werden (Abschnitte 5.4.2 und 5.4.3).

Weil Namen typisiert sind, ist die Verallgemeinerung von Typvariablen zu Variablen für *Namensinstanzen* nicht offensichtlich. Die Bedeutung von Variablen *für Funktionen* wird am Beispiel der Zusicherung einer `totalOrder`-Eigenschaft erläutert (Abschnitt 8.3). Dieses Beispiel unterstreicht die Ausdrucksmächtigkeit algebraischer Spezifikationen, die praktisch nicht gewürdigt wurde und bisher in unzulängliche Sprachimplementierungen mündete.

Durch die Bindung von Typvariablen entstehen polymorphe Funktionen. Die Bindung für Funktionsvariablen ergibt sich erst aus der Analogie zwischen Instanziierung und Funktionsapplikation. Durch die Parametrisierung mit Typen erzielt man Polymorphie und durch die Parametrisierung mit Funktionen entstehen Funktionen höherer Ordnung. Beide Aspekte sind dadurch Teilaspekte der universelleren Generizität durch Parametrisierung!

Funktionen höherer Ordnung

Funktionen höherer Ordnung wurden bisher nur ansatzweise als Parametrisierungsaspekt erkannt. Entweder man beschränkte sich auf Funktionen *erster* Ordnung mit Parametrisierung und erzielte zweite Ordnung, oder man erzielte höhere Ordnung auf zwei Weisen: durch Parametrisierung *und* klassische Funktionen höherer Ordnung (Abschnitt 5.5). Letztere Kluft verschwindet, wenn Namensinstanziierung als Applikation und uninstanziierte Funktionen als Funktionen höherer Ordnung begriffen werden. Damit muss vor allem auch die Instanziierung mit einer uninstanziierten Funktion unterstützt werden. Durch *generischen Import* stehen dann an den *Applikationsstellen* klassische polymorphe Funktionen höherer Ordnung zur Verfügung.

Wenn Instanziierung und Applikation zusammenfallen, dann können auch Namen und Ausdrücke verschmolzen werden. Außerdem werden *Typannotationen* für Namen und Ausdrücke einheitlich (Abschnitt 5.4.1). Die unterschiedliche Notation für die Instanziierung und Applikation kann aus Lesbarkeitsgründen beibehalten werden: nur wenn alle Teile der Applikation Funktionen sind, werden runde Klammern benutzt, ansonsten eckige. Dabei können untergeordnete Typen und Funktionen als Argumente durch *implizite* Parameter vermieden werden (Abschnitt 8.3).

Analogien

Die Analogie zwischen Ausdrücken und Typtermen floss schon in den Sprachentwurf von HASKELL ein. Dort können einheitlich Typ- und Wertkonstruktoren als ungeklammerte Präfixapplikationen notiert werden. Die Gemeinsamkeit beschränkt sich dabei durch die viel einfacheren Typterme allein auf die Parsierung.

Für algebraische Terme ist aber nicht nur die einheitliche Syntax möglich, sondern vor allem die einheitliche Analyse mit *demselben* Algorithmus! Durch diesen Algorithmus werden obige und noch die folgenden Analogien untermauert:

- Polymorphie und Funktionen höherer Ordnung
- unparametrisierte Typen und Konstanten
- Applikationsnotation mit eckigen und runden Klammern
- Namensidentifikation und Typanalyse

Inwieweit λ-, LET- und FIX-Ausdrücke zu Analogien auf der Namensebene führen, müsste noch weiter untersucht werden. LET-Ausdrücke könnten wie Synonyme als Substitutionen verwaltet werden.

Interessant ist noch, dass das Prinzip der inkrementellen Namensraumanalyse auf das parallele LET von OPAL angewendet werden kann. Dort können mehrere nicht-rekursive LET-Gleichungen in beliebiger Reihenfolge angegeben werden.

9.2 Namensraumkonstruktion

Der Namensraum einer Struktur wird etabliert durch Deklarationen, Importe und eine Parametersignatur. Die Namensidentifikation wird an folgenden Stellen mit unterschiedlichen Namensräumen angewendet:

- Typen in Funktionsdeklarationen
- aktuelle Parameter in Importinstanzen
- selektiv zu importierende Namen

- formale Parameter der Parameterliste

Der Namensraum für die Identifizierung von Funktionsdeklarationen und Importinstanzen ist identisch und umfasst genau alle Importe und Deklarationen. Für die Identifikation der selektiv zu importierenden Namen wird der Namensraum der entsprechenden Importinstanz benutzt. Der Namensraum für die Toplevel-Bezeichner formaler Parameter ist auf die Deklarationen beschränkt; für die Identifikation ihrer Typannotationen müssen Importe und die (untergeordneten) Parameter selbst berücksichtigt werden

Die Namensräume müssen nicht transitiv abgeschlossen sein und dieser Vorteil wird am besten durch direkte Importe ausgenutzt. Für optimale Selektionsmöglichkeiten werden verschiedene Notationsvorschläge gemacht:

- selektiver Ausschluss (Abschnitt 7.2.1)
- getrennter Import für Typen und Funktionen (Abschnitt 7.2.2)
- Platzhalter für formale Parameter (Abschnitt 7.2.3)
- eindeutige Importe nur durch Typannotationen (Abschnitt 7.3)

Die Reihenfolge der Deklarationen und der formalen Parameter kann beliebig gewählt werden. Die inkrementelle Namensraumanalyse passt hervorragend zur vorab Berechnung der abgeschlossenen Parametersignatur (Abschnitt 8.2), auch für *implizite* Parameter (Abschnitt 8.3).

Keine Verschattung

Die Beschränkungen sind minimal. An erster Stelle wird Verschattung durch die i.A. nützlichere Überlagerung ersetzt (Abschnitt 3.4). Für die Namensidentifikation wird eine Monotonieeigenschaft angegeben (Abschnitt 6.4), die für die Terminierung der Namensraumanalyse wichtig ist (Abschnitt 8.1.1).

Die einzige Beschränkung der Namensraumanalyse für Reexporte betrifft die scheinbar zyklischen Importe aus Abschnitt 7.4.6. Das Hauptargument gegen Reexporte ist aber, dass sie eine optimale und transparente Namensraumgestaltung praktisch verhindern. Ihr Vorteil für große Namensräume kann besser implizit außerhalb der Sprache oder durch andere Modulkonzepte erzielt werden (Abschnitt 8.4).

Für mehrfach teilinstanziierte Strukturen verbleibt ein potenzielles Überlappungsproblem, das aber durch eindeutig monomorphe Importinstanzen (Abschnitt 7.1) und direkte Importe sicher verhindert wird. Alle Namen sind dann eindeutig polymorph oder überlagert und monomorph. Insgesamt ist es sogar sinnvoll, wenn man diese Einschränkung auf Strukturen ausdehnt. Pro Struktur ist dann zulässig:

- entweder ein generischer Import
- oder nur instanziierte Importe.

Kein Problem hat die Namensraumanalyse mit Wiederholungen (Abschnitt 8.1.3). Der Aufwand für eine echt mächtigere globalere Namensraumanalyse steht in keinem Verhältnis zum Nutzen (Abschnitt 7.1.1). Für die elementaren Eigenschaften – generische Namen und Reihenfolgeunabhängigkeit –, die durch den Sprachentwurf von OPAL angestrebt wurden, ist die lokale Auflösbarkeit ausreichend und sogar besonders angemessen.

9.3 Verwandte Arbeiten

Der vorliegenden Arbeit ging eine Diplomarbeit [Reu98] von Andreas Reuleaux voraus, in der die Signaturanalyse nur für instanziierte Importe untersucht wurde. Dabei sind immer alle Namen monomorph und deswegen auch Reexporte weniger problematisch. Auf die Eindeutigkeit von Importinstanzen wurde dort verzichtet, was mit der iterativen Namensraumkonstruktion zu der problematischen Namensmenge $\texttt{seq}^+[\texttt{nat}]$ führte (Abschnitt 7.1.2). Außerdem ist dort interessant, dass auf explizite instanziierte Importe der global eindeutigen Typkonstruktoren für Tupel $\times$ und Funktionen $\rightarrow$ verzichtet werden konnte, da diese direkt den Typsignaturen entnommen werden können. Zusätzlichen Aufwand bereitete es aber, den Abschluss bezüglich der Instanzen von $\times$ und $\rightarrow$ für *importierte* Funktionen zu bilden.

Im Rahmen dieser Arbeit wurde der OPAL-Parser adaptiert und die Signaturanalyse neu implementiert, aber nicht in das Compiler-Frontend integriert. Tests mit den vorhandenen Quellen des OPAL-Compilers verliefen sehr gut: wenige und kleine, aber zuvor ignorierte Fehler konnten aufgedeckt werden. Aussagekräftig sind diese Tests allerdings nicht, weil die Quellen schon vorher fast alle kontextkorrekt und unkompliziert waren. Vor allem aber hielt sich der Aufwand für die Umstellung der Quellen von Reexporten zu direkten Importen in Grenzen!

In [NN99] wird ein formaler Beweis der polymorphen Typinferenz präsentiert. Dort wird die Konsistenz und Vollständigkeit von Algorithmus $\mathcal{W}$ für Mini-ML mit dem auf ML basierenden System ISABELLE/HOL [Pau94] bewiesen. In Mini-ML gibt es nur den zweistelligen Typkonstruktor für Funktionen; die Notation von LET-Ausdrücken mit de-Bruijn Indizes ist Benutzern so kaum zumutbar. Die Kluft zu ISABELLE mit Typklassen ist insgesamt ziemlich groß, ein *informaler* detaillierter Beweis zum Typklassenkonzept von HASKELL wird in [NN99] als unvollständig erkannt. ISABELLE selbst ist zwar in ML implementiert, aber nicht mit *reinen* Funktionen.

Der formale Beweis von Algorithmus $\mathcal{W}$ stützt sich auf das Unifikationstheorem, wofür auf einen formalen Beweis von Paulson 1985 [Pau85] mit dem LCF-System verwiesen wird. Dort wird auf den beträchtlichen Aufwand (und die Unzulänglichkeiten des Beweissystems) hingewiesen. In Vorarbeiten zu dieser Arbeit wurde der Unifikationsalgorithmus (Abschnitt 2.4) mit umfangreichen Theorien zur *Substitution* und zum *Disagreement Pair* in PVS [OSR93b] spezifiziert und (aus Zeitgründen leider nur *fast* vollständig) bewiesen. Außerdem wurde eine fast lineare Martelli-Montanari-Unifikation [MM82] *rein funktional* in OPAL implementiert, aber eine Integration in die Signaturanalyse nicht konsequent weiter verfolgt.

9.4 Zukünftige Arbeiten

Die hier skizzierte Kontextanalyse bzw. der Algorithmus $\mathcal{I}$ ist nicht leicht in eine relevante Spezifikationssprache wie PVS oder Programmiersprache wie OPAL zu integrieren, da etliche Quellen adaptiert werden müssten. PVS unterstützt ein Subtypkonzept und automatische Typkonversionen, deren orthogonale Integration hier nicht untersucht wurde.

OPAL erlaubt eine flexible, Klammern sparende Mixfix-Notation mit Präzedenzen und Assoziativitäten, d.h. elementare Funktionsapplikationen sind syntaktisch nicht eindeutig erkennbar und müssen durch die Typanalyse bestimmt werden. Eine Mixfixnotation ist nicht nur für Funktionsapplikationen, sondern auch für Namensinstanziierungen wünschenswert. Schon die Tupel- und Funktionstypkonstruktoren werden als spezielle Mixfixe notiert, es fehlt also nur noch eine Verallgemeinerung auf benutzerdefinierte und potenziell überlagerte Namen. Möglicherweise verletzen aber Präzedenzen und Assoziativitäten wie die Verschattung sogar die für die Namensraumanalyse nötige Monotonieeigenschaft (Abschnitt 6.4). Gerade für Mixfix-Ausdrücke ist dann die Trennung von Namensraumkonstruktion und Typanalyse wichtig, wie sie durch obligatorische Funktionsdeklarationen gewährleistet wird.

Für die hier skizzierte Kontextanalyse ist ein formaler Korrektheitsbeweis mit einem Beweissystem wie PVS vorstellbar. Insgesamt könnte ein algebraischer Sprachentwurf angestrebt werden, mächtig genug, um ein Beweissystem wie PVS und einen Übersetzer wie OPAL zu spezifizieren und zu implementieren, und insbesondere den Beweis der zugehörigen Kontextanalyse zu ermöglichen. Eine Erweiterung von OPAL um ein Beweissystem mit zugehörigen Sprachelementen wird in [Did97] beschrieben.

Ein wichtiger Aspekt für die Entwicklung korrekter Programme ist die automatische Generierung von Standardfunktionen für benutzerdefinierte Datentypen, wie das in HASKELL durch Ableiten (*Deriving*) z.B. von Gleichheiten oder Ordnungen möglich ist. In PVS werden automatisch `map`- und `reduce`-Funktionen zu parametrisierten und rekursiven Datentypen erzeugt. Die generische Programmierung derartiger Funktionen durch den Benutzer, in Abhängigkeit vom Aufbau eines freien Datentyps, erlauben die so genannten *polytypic* Sprachen [Hin99].

Außer nur die Korrektheit von Programmen zu beweisen, könnte analog zum Terminierungsmaß von PVS auch eine formale Abschätzung der Größenordnung von Laufzeit und Speicherbedarf in ein Beweissystem integriert werden. Mit einer zuverlässigen *worst-case* Prognose über den Ressourcenbedarf kann der Absturz eines Programms durch Ressourcenerschöpfung ausgeschlossen werden. Ist der Ressourcenbedarf für den Übersetzer und Beweiser selbst in Abhängigkeit von Quelltextgröße vorhersehbar, dann ist eine sichere Prüfung und Übersetzung von Quelltexten zusammen mit einer unmittelbar anschließenden Ausführung möglich. In diesem Fall wäre die fast-lineare Unifikation sowie eine insgesamt polynomiale Namensidentifikation $\mathcal{I}$ interessant.

Literaturverzeichnis

[ACF93] L. Albert, R. Casas, and F. Fages. Average-case analysis of unification algorithms. *Theoretical Computer Science*, 113(1):3–34, May 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[B+93] Manfred Broy et al. The Requirement and Design Specification Language SPECTRUM— An Informal Introduction. (version 1.0), March 1993.

[Bar84] H.P. Barendregt, editor. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984. (revised edition).

[Bar91] H.P. Barendregt. Lambda Calculi with Types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford Science Publications, 1991.

[BBD+81] F.L. Bauer, M. Broy, W. Dosch, R. Gnatz, B. Krieg-Brückner, A. Laut, M. Luckmann, T.A. Matzner, B. Möller, H. Partsch, P. Pepper, K. Samelson, R. Steinbrüggen, M. Wirsing, and H. Wössner. Programming in a wide spectrum language: a collection of examples. *Science of Computer Programming*, 1:73–114, 1981.

[BDDG93] Ralph Betschko, Sabine Dick, Klaus Didrich, and Wolfgang Grieskamp. Formal Development of an Efficient Implementation of a Lexical Scanner within the KORSO Methodology Framework. Technical Report 93-30, TU Berlin, October 1993.

[BER94] Didier Bert, Rachid Echahed, and Jean-Claude Reynaud. *Reference Manual of the* LPG *Specification Language and Environment*. IMAG, Grenoble, June 1994. Release with disequations.

[BGJ89] R. S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[BJ95] Manfred Broy and Stefan Jähnichen, editors. KORSO*: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*. Springer, Heidelberg, November 1995.

[BM00] Michel Bidoit and Peter D. Mosses. A gentle introduction to CASL. Tutorial, CoFI Workshop at the 3rd European Joint Conferences on Theory and Practice of Software (ETAPS'2000), Berlin, Germany, April 2000.

[CB83] J. Corbin and M. Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In R. E. A. Mason, editor, *Proceedings of the IFIP'83 Ninth World Computer Congress*, pages 909–914, Paris, September 1983. North-Holland.

[CF99] Carlos Camarão and Lucília Figueiredo. Type Inference for Overloading without Restrictions, Declarations or Annotations. In *FLOPS '99, Tsukuba, Japan*, volume 1722 of *LNCS*, pages 37–52. Springer, November 1999.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[dC86] Dennis de Champeaux. About the Paterson-Wegman Linear Unification Algorithm. *Journal of Computer and System Sciences*, 32(1):79–90, February 1986.

[DEGP94] Klaus Didrich, Jürgen Exner, Wolfgang Grieskamp, and Peter Pepper. Integrating Algebraic Specifications and Functional Programming. Experiences with the OPAL System. In *Proceedings of the International Workshop on Advanced Software Technology, Shanghai*, September 1994.

[DFG+94] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland*, volume 782 of *LNCS*, pages 228–244. Springer, March 1994.

[DGG+96] Klaus Didrich, Carola Gerke, Wolfgang Grieskamp, Christian Maeder, and Peter Pepper. Towards Integrating Algebraic Programming and Functional Programming: the OPAL System. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 559–562. Springer, 1996.

[DGMP97] Klaus Didrich, Wolfgang Grieskamp, Christian Maeder, and Peter Pepper. Programming in the Large: the Algebraic-Functional Language OPAL 2α. In *Proceedings of IFL '97: 9th International Workshop on Implementation of Functional Languages, St Andrews, Scotland*, volume 1467 of *LNCS*, pages 323–338. Springer, September 1997.

[Did97] Klaus Didrich. Compiler Support for Correctness Proofs. In *Automated Theorem Proving in Software Engineering (CADE-14 workshop)*, 1997.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of POPL'82*, pages 207–212. ACM, 1982.

[EFP94] Gottfried Egger, Andreas Fett, and Peter Pepper. Formal Specification of a Safe PLC Language and its Compiler. In Victor Maggioli, editor, *SAFECOMP'94, Proceedings of the 13th International Conference on Computer Safety, Reliability and Security Anaheim, Kalifornien, USA*, pages 11–20, October 1994.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1990.

[Exn94] Jürgen Exner. The OPAL Tutorial. Technical Report 94-9, TU Berlin, May 1994.

[FGMO87] K. Futatsugi, J. Goguen, J. Meseguer, and K. Okada. Parameterized Programming in OBJ2. In *Proceedings of the 9th In-*

ternational Conference on Software Engineering, pages 51–60, Monterey, CA, March 1987. IEEE Computer Society Press.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. International Computer Science Series. Addison-Wesley, 1988.

[Gog84] Joseph A. Goguen. Parameterized Programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gro94] The Opal Group. The Programming Language Opal. Internal report, 4th edition, 1994.

[GW88] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 1988.

[GWM+93] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, October 1993.

[Han99] Keith Hanna. Implementing theorem provers in a purely functional style. *Journal of Functional Programming*, 9(2):147–166, March 1999.

[Hen91] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, December 1991.

[Hin99] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3:193–214, 1999.

[HM94] Fritz Henglein and Harry G. Mairson. The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming*, 4(4):435–477, October 1994.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice Hall, London, 1987.

[KHM94] Paris C. Kanellakis, Gerd G. Hillebrand, and Harry G. Mairson. An Analysis of the Core-ml Language: Expressive Power

and Type Reconstruction. Technical Report CS-94-25, Brown University, May 1994.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, May 2000.

[LY98] Oukseh Lee and Kangkeun Yi. Proofs about a Folklore LET-Polymorphic Type Inference Algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[MF91] David A. Mundie and David A. Fisher. Optimized Overload Resolution and Type Matching for ADA. *ACM SIGADA ADA Letters*, 11(3):83–90, Spring 1991.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MM82] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MTH90] R. Milner, M. Tofke, and R. Harper. *The Definition of Standard* ML. Mass. Institute of Technology Press, Cambridge, Mass., 1990.

[MW81] Zohar Manna and Richard Waldinger. Deductive Synthesis of the Unification Algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *6th International Conference on Programming Languages*, volume 167 of *LNCS*. Springer, 1984.

[Naz95] Dieter Nazareth. *A Polymorphic Sort System for Axiomatic Specification Languages*. PhD thesis, Technische Universität München, May 1995.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type Inference Verified: Algorithm W in ISABELLE/HOL. *Journal of Automated Reasoning*, 1999.

[OS97a] Sam Owre and Natarajan Shankar. Abstract Datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1997.

[OS97b] Sam Owre and Natarajan Shankar. The Formal Semantics of PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *The* PVS *Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the* PVS *Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. 3 volumes: Language, System, and Prover Reference Manuals.

[Pau85] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–169, 1985.

[Pau94] L.C. Paulson. ISABELLE*: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pau96] L.C. Paulson. ML *for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[PDM80] T. Pennello, F. DeRemer, and R. Meyers. A Simplified Operator Identification Scheme for ADA. *ACM SIGPLAN Notices*, 15(7–8):82–87, July–August 1980.

[Pep98] Peter Pepper. *Funktionale Programmierung in* OPAL*,* ML*,* HASKELL *und* GOFER. Springer, 1998. Lehrbuch.

[PH+97] John Peterson, Kevin Hammond, et al. Report on the programming language HASKELL, a non-strict, purely functional language (version 1.4). Technical report, Yale University, April 1997. HASKELL committee.

[PS96] Peter Pepper and Douglas R. Smith. A High-Level Derivation of Global Search Algorithms (with Constraint Propagation). *Science of Computer Programming*, 1996.

[PW78] M. S. Paterson and M. N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[PW94] Peter Pepper and Martin Wirsing. KORSO: A Methodology for the Development of Correct Software. Technical Report 94-36, TU Berlin, November 1994.

[Reu98] Andreas Reuleaux. Formale Spezifikation eines Analysealgorithmus für Namensräume. Diplomarbeit, TU-Berlin, September 1998.

[Rob71] J. A. Robinson. Computational Logic: The Unification Computation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 63–72. Edinburgh University Press, Edinburgh, Scotland, 1971.

[RP89] Peter Ruzicka and Igor Prívara. An Almost Linear Robinson Unification Algorithm. *Acta Informatica*, 27(1):61–71, 1989.

[Sch95] Michael I. Schwartzbach. Polymorphic Type Inference. Lecture Series LS-95-3, BRICS, University of Aarhus, June 1995. viii+24 pp.

[Smi90] Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Smi91] Geoffrey S. Smih. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, August 1991.

[SP93] Douglas R. Smith and Eduardo A. Parra. Transformational Approach to Transportation Scheduling. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference, Chicago, IL*, pages 60–68, 1993.

[Thi94] Peter Thiemann. *Grundlagen der funktionalen Programmierung*. B. G. Teubner Stuttgart, 1994.

[Tho96] Simon Thompson. HASKELL*: The Craft of Functional Programming*. Addison Wesley, 1996.

[VS91] Dennis M. Volpano and Geoffrey S. Smith. On the Complexity of ML Typability with Overloading. Technical Report TR91-1210, Cornell University, May 1991.

[W+92] Martin Wirsing et al. A Framework for Software Development in KORSO. Technical Report 9205, Ludwig-Maximilians-Universität München, 1992.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms*. Series in Computer Science. Prentice Hall International, 1990.

[WB89] Philip Wadler and Stephen Blott. How to make ad–hoc polymorphism less ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, January 1989.

[WDC+95] Uwe Wolter, Klaus Didrich, Felix Cornelius, Markus Klar, Roland Wessäly, and Hartmut Ehrig. How to cope with the spectrum of SPECTRUM. In Manfred Broy and Stefan Jähnichen, editors, KORSO*: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*. Springer, 1995.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer, 1984.

[Wir85] Niklaus Wirth. *Programming in* MODULA-*2*. Springer, 3rd edition, 1985.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung*. Springer, 1992. Lehrbuch.

[WS80] Peter J. L. Wallis and Bernard W. Silverman. Efficient Implementation of the ADA Overloading Rules. *Information Processing Letters*, 10(3):120–123, April 1980.